X/O/P/E/N/

PORTABILITY GUIDE
PROGRAMMING LANGUAGES

4

X/O/P/E/N/

PORTABILITY GUIDE

PROGRAMMING LANGUAGES

X/OPEN

PORTABILITY GUIDE

PROGRAMMING LANGUAGES

X/OPEN PORTABILITY GUIDE

Set of 5 Volumes		ISBN: 0-444-70179-6
Volume 1	XVS Commands and Utilities	ISBN: 0-444-70174-5
Volume 2	XVS System Calls and Libraries	ISBN: 0-444-70175-3
Volume 3	XVS Supplementary Definitions	ISBN: 0-444-70176-1
Volume 4	Programming Languages	ISBN: 0-444-70177-X
Volume 5	Data Management	ISBN: 0-444-70178-8

Published by:

ELSEVIER SCIENCE PUBLISHERS B.V.
P.O Box 1991
1000 BZ Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Any comments relating to the material contained in the X/OPEN Portability Guide may be submitted to the X/OPEN Group by letter via the Publisher or directly by Electronic Mail to:

xopen@inset.co.uk

PRINTED IN THE NETHERLANDS

Contents

Trademarks

UNIX™ is a registered trademark of AT&T in the USA and other countries.

C-ISAM™ is a trademark of Informix Corporation.

LEVEL II COBOL™ is a trademark of Micro Focus Limited.

XENIX™ is a trademark of Microsoft Inc.

IBM™ is a trademark of International Business Machines Corp.

X/OPEN™ is a licensed trademark of the X/OPEN Group Members.

POSIX™ is a trademark of the Institute of Electrical and Electronic Engineers Inc.

Preface

X/OPEN represents a major breakthrough in the world of standards for the information technology industry. Ten* of the world's major information system suppliers have come together to encourage applications portability resulting in tangible benefits for computer users, independent software houses and for the suppliers themselves.

The Group's principal aim is to increase the volume of applications available and to maximise the return on investments in software development made by Users and Independent Software Vendors.

This is achieved by ensuring portability of application programs at the source code level. Through this portability, users can mix and match computer systems and applications software from many suppliers, and thus investment in applications software is protected into the future.

In order to provide such portability, the Group defines a **Common Applications Environment** built on the interfaces to the **UNIX** operating system, as defined in the AT&T System V Interface Definition, and covering other aspects required of a comprehensive applications interface.

The X/OPEN Portability Guide contains an evolving portfolio of practical standards for application portability. All of the members of X/OPEN guarantee to support the standards defined, leading to:

- Growing portability
- No dependence on a single source - freedom of choice
- Increased application software selection
- More security in software investments
- International support for the *Common Applications Environment*

X/OPEN is not a standards-setting organisation; it is a joint initiative by members of the business community to integrate evolving standards into a common, beneficial and continuing strategy.

---

* At the time of publication, the membership of the X/OPEN Group was BULL, DEC, ERICSSON, HEWLETT-PACKARD, ICL, NIXDORF, OLIVETTI, PHILIPS, SIEMENS and UNISYS

Issue 2 of the X/OPEN Portability Guide (published in January 1987) comprises five volumes defining the interfaces currently identified as components of the Common Applications Environment.

Volume 1	System V Specification: Commands and Utilities
Volume 2	System V Specification: System Calls and Libraries
Volume 3	System V Specification: Supplementary Definitions XVS Internationalisation XVS Terminal Interfaces XVS Inter-Process Communication XVS Source Code Transfer
Volume 4	Programming Languages C Language COBOL Language
Volume 5	Data Management Indexed Sequential Access Method (ISAM) Relational Database Language (SQL)

In addition, each volume includes an introduction giving the philosophy of the Common Applications Environment and an overview of its components.

This guide is aimed at both the decision makers and the implementation teams of:

- Independent Software Vendors
- Software Houses
- Users
- Equipment Manufacturers

The Guide is designed to sit permanently on the desk, serving as a common reference point for anyone directly concerned with the practical side of software development, namely systems designers, programmers and consultants.

The various parts of the Portability Guide are closely interrelated. Any reference from one part of the definition to another part uses the title of the other part as a reference (e.g., "XVS TERMINAL INTERFACES").

Acknowledgements

X/OPEN gratefully acknowledges:

- **AT&T** for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The **/usr/group** Standards Committee, whose Standard contributed to the Group's work.
- **Informix Corporation.** of Menlo Park, California (Telex no. 361834) for permission to use material from the specification of their C-ISAM product and for provision of that material in machine readable form.
- **Micro Focus Ltd.** of Newbury, Berkshire for permission to use material from the specification of their LEVEL II COBOL compiler.
- The assistance given by the following companies in the preparation of the Database Language (SQL) definition:

  Informix Corporation
  Oracle Corporation
  Queensland Information Technology
  Relational Technology Inc.
  Unify Corporation

Referenced Documents

The following documents are referenced in this guide:

- System V Interface Definition (Spring 1985 - Issue 1)
- System V Interface Definition (Spring 1986 - Issue 2)
- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2)
- UNIX System V - Release 2.0 Programming Guide (April 1984 - Issue 2)
- ANS Draft Proposal for C Language (October 1986 - ANS X3J11/86-151)
- 1984 /usr/group Standard
- IEEE P1003.1 Trial Use Standard (April 1986)
- Informix Corporation C-ISAM Reference Manual (Version 2.10 - January 1985)
- MicroFocus Level II COBOL Language Reference Manual (Version 2.5 and 2.6, Issue 7 - April 1984)
- Standard for COBOL (ANS X3.23-1974)
- Standard for COBOL (ANS X3.23-1985)
- Standard for FORTRAN (ANS X3.9-1978)
- Standard for Database Language (SQL) (ANS X3.135-1986)
- Standard for PASCAL (ISO 7185-1983)

X/OPEN

PORTABILITY GUIDE

THE COMMON APPLICATIONS ENVIRONMENT

Contents

Chapter 1

The Common Applications Environment

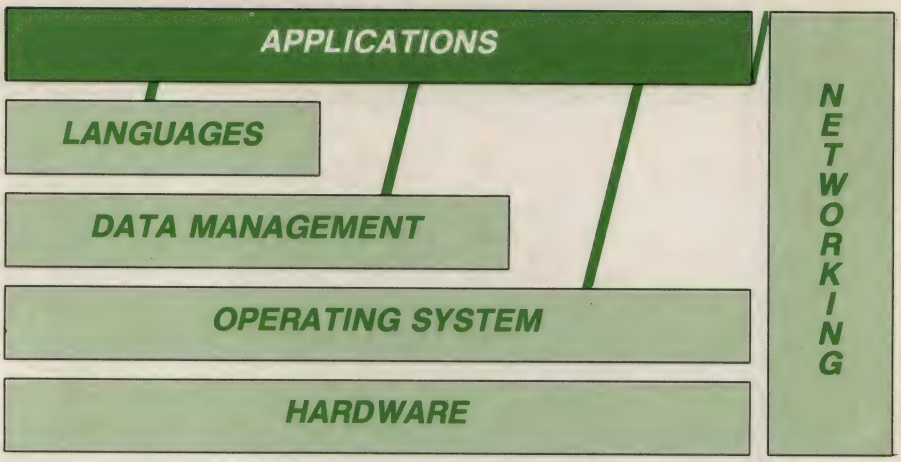
The formation of the X/OPEN Group represents a major initiative by an international group of suppliers of computer systems to create a free and open market, offering Independent Software Vendors (ISVs) as wide a market as possible for their products and giving users an increased return on investment in application software.

The current dominance of proprietary machine environments is restricting the growth of the computer industry. Users tend to get locked into a particular proprietary system by the investment they have made in the applications. Independent Software Vendors are discouraged from writing applications for a particular environment because of the limited markets caused by this fragmentation. This means that there is very little generally available software for each type of system, thus increasing the size of investment needed by each user. All this in turn limits the sales potential of machines from the computer suppliers.

The objective shared by the members of the X/OPEN Group is to establish a Common Applications Environment to the mutual advantage of users, Independent Software Vendors and computer suppliers. Applications written to operate in this environment will be portable at the source code level to a wide range of machines, thereby releasing the user from dependence on a single supplier, reducing the necessary investment in applications, considerably increasing the market for independent software and opening up the market for systems suppliers.

The existence of these "Open Systems" allows users to mix and match systems from different suppliers, and to move applications between machines to meet changing requirements as business grows, thereby giving protection of investment in applications software into the future.

The great increase in the potential market encourages the Independent Software Vendors to produce a wealth of general applications packages, and the availability of this further reduces the investment needed by the users. The whole situation is thus mutually reinforcing.



The foundations of the Common Applications Environment are the interfaces of the UNIX System V operating system, as defined in the AT&T "System V Interface Definition", and the C language.

To define a complete environment for portable applications, it is also necessary to satisfy the requirements for data management, integration of applications, data communications, distributed systems, the use of high level languages and the many other aspects involved in providing a comprehensive applications interface. The X/OPEN Group intends, therefore, to publish progressively definitions covering these areas.

The systems of the X/OPEN Group members that support interfaces derived from UNIX operating systems will do this according to the X/OPEN definitions and will support the full Common Applications Environment.

A specific Common Applications Environment feature may not, however, be present if it is not relevant in the market area in which a particular system is sold. For example, a system sold only in a scientific context might not support COBOL. Conversely, a particular system may support features over and above those of the Common Applications Environment, some of which may partially overlap. An example of this could be that an alternative dialect of COBOL is supported in addition to that of the Common Applications Environment.

The X/OPEN Group is primarily concerned with standards selection and adoption. The general policy is to use International Standards, where they exist, and to adopt "de facto" standards in other cases.

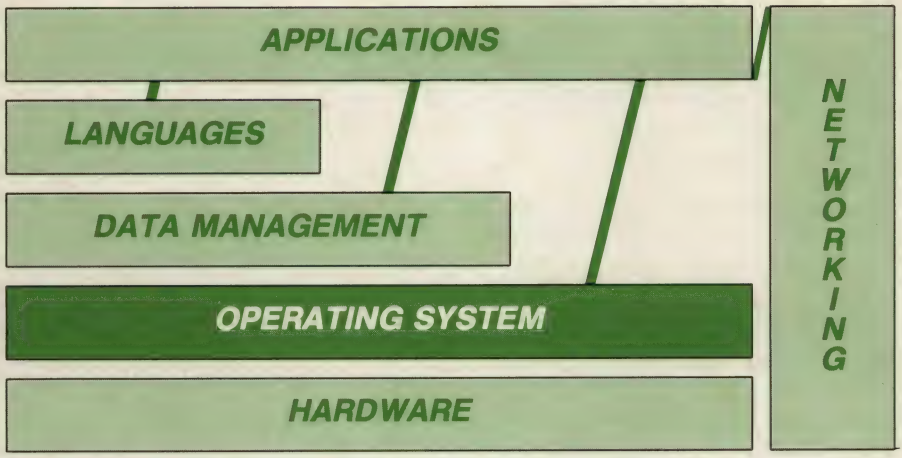
Where International Standards do not exist, it is X/OPEN policy to work closely with standardisation bodies to encourage their emergence.

It is important that the defined elements of the Common Applications Environment be readily achievable on member systems, and have wide acceptance. For this reason, the definitions, in general, fall within the capabilities of at least one currently available popular product.

In this guide, certain aspects of the Common Applications Environment are defined with reference to the interfaces offered by specific products. This does not mean that member systems will necessarily contain these products, but that the defined interfaces will be supported. Indeed the method of support for an interface on a particular system may change with time.

Chapter 2

X/OPEN System V Specification (XVS)



2.1 INTRODUCTION

The X/OPEN System V specification (XVS) defines the applications interfaces provided by the underlying operating system and forms the foundation of the Common Applications Environment.

The XVS is derived from a series of standards activities. The evolving standard is briefly addressed, and the relationship between the X/OPEN System V Specification and the System V Interface Definition and other standards is explained.

2.2 THE EVOLVING STANDARD

2.2.1 Origins

The UNIX operating system was developed by Ritchie and Thompson at Bell Laboratories in the early 1970s. The current AT&T System V version may be traced back directly to that first system.

For many years, it remained basically an academic product. More recently, computer suppliers have adopted the UNIX system as a multi-tasking, multi-user and portable operating environment. They have based their systems on one of several releases, variants or look-alikes. Of these, the most widely used were Version 7, System III, the Berkeley system and XENIX.

Although these systems had much in common, the degree of compatibility at the application interface level was insufficient to permit the development of totally portable applications.

2.2.2 The IEEE "POSIX" Standard

/usr/group, a group of users of UNIX derivatives in the USA, established a committee with the objective of proposing a set of standards for application level interfaces. After publishing its standard, together with a reviewer's guide, the group decided to seek IEEE status for the standard. In late 1984, the /usr/group standards committee closed its activities in its own name and its members were encouraged to become involved in the IEEE group, known as P1003.

The P1003 group published a "trial-use" standard in early 1986, which has the status of a "Draft American National Standard". This "Portable Operating System for Computer Environments" (POSIX) is expected to be revised and submitted for approval in 1987.

The IEEE P1003 group is working to extend the POSIX standard. It is expected that the next area to be standardised will be the subset of commands which offer an interface to applications.

2.2.3 The AT&T System V Interface Definition

The "System V Interface Definition" (SVID), first published by AT&T in the spring of 1985, represented a major standards initiative. AT&T were prominent in the activities of /usr/group and the influence of /usr/group can clearly be seen in the SVID. The stated purpose of the SVID is to define common interfaces for all System V implementations.

The definition groups interfaces into a mandatory *base* plus a series of *extensions*. The *base* interfaces must be present in any implementations of System V. If any interface from an *extension* is supported, it must adhere to the definition.

Issue 1 of the SVID comprised a single volume defining operating system interfaces (known as *system calls* and *library routines*) available to applications as directly called external functions and defined in terms of invocation from C-language programs.

Issue 2 of the SVID was published in early 1986 and comprised two volumes. The first volume contained the same material as issue 1, with some restructuring to improve ease of use and some changes to correct errors. It comprised the *Base System Definition* plus a single extension referred to as the *Kernel Extension*.

The second volume primarily defined commands and utilities, normally invoked through a command interpreter. It comprised further extensions referred to as the *Base Utilities Extension*, *Advanced Utilities Extension*, *Administered Systems Extension*, *Software Development Extension*, and *Terminal Interface Extension*. The latter two include library routines in addition to utilities.

2.3 THE X/OPEN SYSTEM V SPECIFICATION

The X/OPEN System V Specification (XVS) is based upon the AT&T System V Interface Definition, but also taking into account the trial use standard published by IEEE and the capabilities of the existing AT&T System V product.

The XVS is organised into a number of self-contained sections:

"XVS SYSTEM CALLS AND LIBRARIES" defines the Operating System Interfaces and broadly corresponds to Volume 1 of the SVID.

"XVS COMMANDS AND UTILITIES" defines commands and utilities and broadly corresponds to Volume 2 of the SVID. The purpose of the X/OPEN Portability Guide is to facilitate the portability of applications. As such, system administration is outside of its scope and the routines included in the AT&T *Administered System Extension* are not defined.

"XVS TERMINAL INTERFACES" defines a set of portable interfaces to locally connected asynchronous terminals and broadly corresponds to the AT&T Terminal Interface Extension in Volume 2 of the SVID.

"XVS INTER-PROCESS COMMUNICATION" defines interfaces to shared memory, semaphores and message passing, included as an interim mechanism to satisfy the immediate requirements of Inter-Process Communication facilities.

2.3.1 System Calls and Libraries

"XVS SYSTEMS CALLS AND LIBRARIES" contains a full definition of interfaces to system calls and library routines and broadly corresponds to Volume 1 of the SVID.

The X/OPEN Group has extended the SVID in a number of areas:

- Certain changes have been included, which the SVID denotes as future directions.
- The use of symbolic names to replace numeric constants, introduced by AT&T in their SVID, has been extended.
- Clarification of existing wording has been introduced in a limited number of places to "tighten" the specification.
- The opportunity has been taken to correct clerical errors in the SVID.
- Definitions have been included of a number of further UNIX System V Release 2.0 functions which are in widespread use by application developers.

The relationship between the XVS and the SVID is clearly stated. The whole of the SVID *base* definition is included as mandatory with the exception of the maths group, which is not mandatory for systems sold into markets where it is not relevant. *Termio* was not mandatory in issue 1 of the XVS because of some difficulties in implementation. These have now been resolved, and the routine is now mandatory, except in systems which do not support locally connected asynchronous lines.

The XVS incorporates all the interfaces within the SVID *kernel extension set* although a number are defined as optional.

In the XVS, interfaces defined as *optional* will be available on most but not necessarily all X/OPEN systems; use of them could restrict portability. Any *optional* interface supported on an X/OPEN system will conform to the X/OPEN specification.

The XVS defines interfaces in terms of their interface syntax and run-time behaviour, without constraining the method of their implementation. The names "system calls" and "subroutines" are retained purely for compatibility with other documentation.

2.3.2 Inter-process Communication

The kernel extension interfaces relating to shared memory, semaphores and message passing are included in "XVS INTER-PROCESS COMMUNICATION" as a short-term mechanism to satisfy the immediate requirements for Inter-Process Communication facilities. However, they are machine specific and cannot be supported on all hardware architectures. The Group believes that a more generalised approach to the whole subject of Inter-Process Communication is required.

2.3.3 Commands and Utilities

"XVS COMMANDS AND UTILITIES" contains a full definition of interfaces to commands and utilities and broadly corresponds to Volume 2 of the SVID. The interfaces are split functionally into those which are intended to provide an applications interface (referred to as *Standard Utilities*) and those which are only intended to be used by development programmers or during the porting of applications to an X/OPEN system (referred to as *Development Utilities*). The Standard Utilities will be present in all X/OPEN systems, as they are needed to provide a run-time interface to applications. The Development Utilities may only be present in development systems; their respective descriptions are clearly annotated to indicate this. This same distinction is also present in the SVID. The X/OPEN development utilities correspond exactly to the SVID *Software Development Extension*.

The definition of standards for commands and utilities is an evolving process and X/OPEN intends to participate fully.

The current definition is a valuable first step, but additional work will be needed to evolve towards a complete standard. For example:

- The number of options defined for many of the commands is excessive and includes functionality which is rarely used or is implementation specific.
- There are too many different ways of achieving the same results.
- Many of the current descriptions were written to record the observed behaviour of already existing utilities and the level of precision is inadequate for use as a definitive standard.

Rectification of this is an enormous task and the current X/OPEN definition is of necessity based on the available documentation, but incorporates extensive annotation to highlight potential portability problems resulting from the points listed above. To achieve maximum portability, application developers should avoid the use of the functionality so annotated.

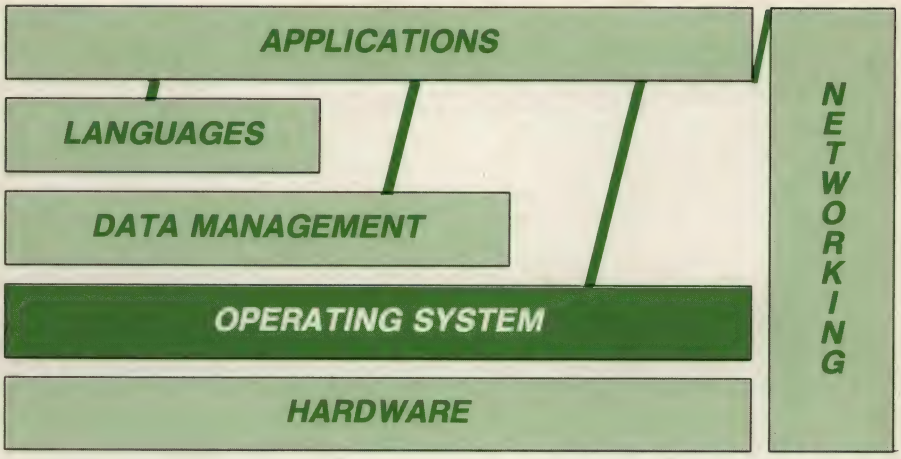
X/OPEN is working on a substantially improved definition of commands, with the number of options reduced to those in common use and with a higher level of specification. In addition to the current X/OPEN specification, "XVS COMMANDS AND UTILITIES" contains a proposed template for improved specifications together with a number of examples.

This improvement process will be carried out in close consultation with the various user organisations and standards bodies, such as IEEE and ISO, to ensure that the result is a single standard definition of operating system commands and utilities.

Readers of the X/OPEN Portability Guide are invited to participate directly in the consultative process to ensure that the evolving standard matches the requirements of existing and future applications.

Chapter 3

Internationalisation



3.1 INTRODUCTION

X/OPEN members market systems in many countries. Our customers and users speak many different languages and conform to different cultural conventions and business practices. It is important therefore that X/OPEN systems are capable of supporting a range of language and cultural environments. In many cases a strong requirement also exists to cope with these variations on the same system. An example is within the administration of the European Economic Community.

To date UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set and on American English. There are no facilities for dealing with other coded character sets, nor for supporting different languages and cultural conventions.

The requirement for effective mixed language working brings with it the need for coded character sets larger than can be accommodated by 7-bit characters, as does the requirement to support the more complex languages. At the same time there is a trade-off between the ability to handle larger coded character sets, and the amount of storage required to hold the data. For most European requirements an 8-bit system provides the correct balance. For the major Eastern languages (such as Chinese and Japanese) a 16-bit system is necessary, even to support a single language.

To satisfy these requirements enhancements must be made to the system to provide full data transparency to applications, allowing flexiblity in the choice of coded character set(s) employed. Additionally, the system must allow program messages (both input and output) to be handled in the native language of each user, as well as providing cultural dependent data items (such as date formats and currency symbols).

3.2 THE X/OPEN NATIVE LANGUAGE SYSTEM

The X/OPEN Native Language System (NLS) is a set of interfaces designed to facilitate the development of applications that can operate in many different language and cultural environments. The interfaces have been derived from those of the Native Language Support system developed by the Hewlett-Packard Company of Palo Alto, California. They have been further enhanced by X/OPEN and have been modified in strategic areas to more closely relate to the Internationalisation proposals of the Draft Proposed American National Standard for the C Programming Language.

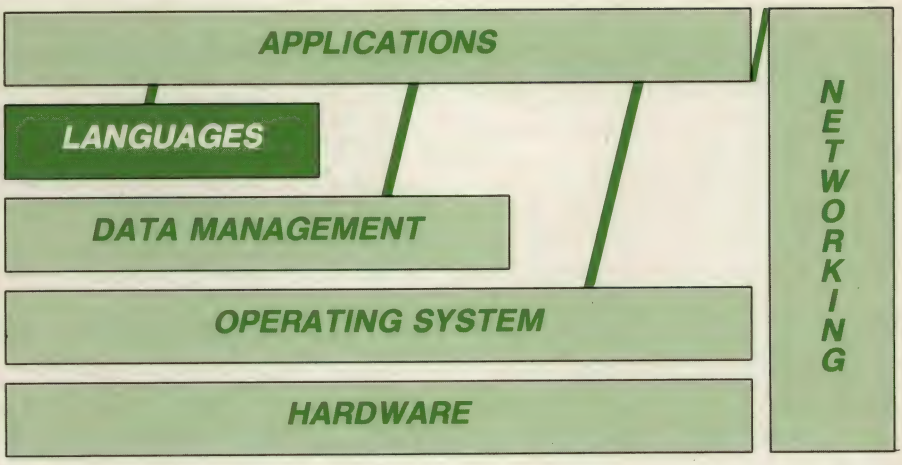
The first issue of the specification, defined in "XVS INTERNATIONALISATION", concentrates on facilities for the development of internationalised applications (rather than on internationalising the operating system itself), and on the 8-bit coded character set situations.

The following groups of facilities are defined:

- A message catalogue system which allows program messages to be held apart from the program logic, translated into different native languages, and the appropriate version retrieved by the program at run time.
- An announcement mechanism whereby native language, local custom (territory) and codeset requirements appropriate to each user can be identified to applications at run time.
- Enhanced interface definitions of standard C library functions, which provide language dependent character type classification, upper to lower case and lower to upper case character conversions, date and time messages, floating point to string conversions, and text collation.
- Library functions which allow programs to determine cultural and language specific data dynamically (e.g. the format of date and time strings, weekday and month names, currency symbols, etc.).
- A set of standard commands and library functions which will operate correctly with 8-bit characters.

Chapter 4

C Language



4.1 INTRODUCTION

This chapter addresses the C language and guidelines for portability when writing C code.

Currently the American National C Language Standards committee, X3J11, is working towards a standard for the C programming language. The X/OPEN Group is represented on that committee by member companies and intends to adopt the standard, once it has been established as a practical reality.

Meanwhile, the X/OPEN definition included in "C LANGUAGE" is based upon that given in Chapter 2 of the "System V Programming Guide", Release 2.0, published by AT&T.

4.2 C LANGUAGE PORTABILITY GUIDELINES

Whilst the C language provides the basis for applications portability, it is easy to write statements, using valid C constructs, that are machine specific. Care has to be taken when writing programs that are intended to be portable across a range of systems. "C LANGUAGE" includes advice towards ensuring portability.

4.3 THE ANS X3J11 DRAFT STANDARD

The ANS X3J11 standard has been published in a draft form but may still change before it is approved. However, it is already clear that the standard will impose certain restrictions such that programs written to the current C language definition may not work correctly, if the source is later passed through a compiler that supports the ANS standard.

To address this, "C LANGUAGE" includes advice on writing programs to avoid these problems.

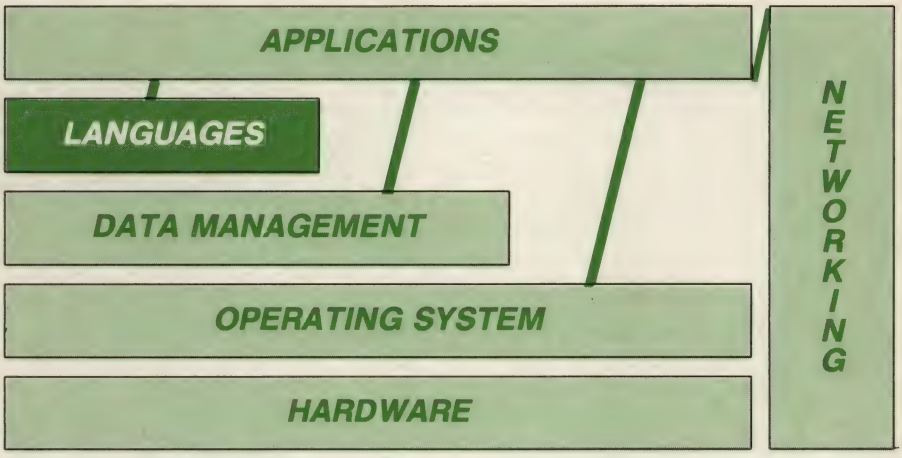
4.4 THE C PROGRAM PORTABILITY CHECKER (*lint*)

The *lint* program checks C source programs for violation of most of the portability rules. It also gives a more stringent enforcement of the type rules of C than is provided by most C compilers. A further option detects a number of wasteful or error-prone constructions which nevertheless are syntactically correct.

Use of the *lint* program is recommended: it is described in "C LANGUAGE".

Chapter 5

Other Programming Languages



5.1 INTRODUCTION

This chapter addresses programming languages other than C on X/OPEN systems. It covers the inclusion of the principal high level languages in the Common Applications Environment.

To date, The X/OPEN Group has established definitions for COBOL and FORTRAN and PASCAL.

5.2 COBOL

The X/OPEN COBOL definition identifies a common set of language facilities that will be supported by COBOL compilers on all member systems. Applications written to this definition will be portable to any X/OPEN system.

The ISO Working Group and ANS COBOL committee have been working for some years towards a revised standard for COBOL to reflect more accurately the capabilities of modern COBOL compiling systems. The latest international standard, "X3.23 - 1985" was approved during 1985. At the time of publication of Issue 2 of the Portability Guide, there are few compilers in compliance with the revised standard and hence the X/OPEN group feels that major changes in the X/OPEN definition to reflect the new standard would be premature. It is likely, however, that any future edition of the X/OPEN COBOL language definition will relate to "COBOL 1985" and the new standard has been used as a reference when eliminating obsolete elements from the COBOL definition.

The most widely followed standard for COBOL is still that defined in the earlier 1974 Standard, "ANS X3.23-1974", to which most current COBOL compilers substantially conform.

The 1974 standard is incomplete in the area of facilities for interaction with the on-line user. To overcome this deficiency, most COBOL compilers provide extensions to the *ACCEPT* and *DISPLAY* verbs, but they do this in incompatible ways. Since the majority of applications now include interactive operation, it is necessary for a standard form of *ACCEPT* and *DISPLAY* to be defined in the X/OPEN Common Applications Environment.

In order to have an X/OPEN definition that is achievable on member systems within a short timescale, and one that would have immediate widespread acceptance, it has been based on the definition of COBOL embodied in a popular product: Micro Focus LEVEL II COBOL, which itself conforms to the "ANS X3.23-1974".

The Micro Focus LEVEL II COBOL language specification includes a number of other extensions beyond the 1974 standard, in addition to those to *ACCEPT* and *DISPLAY*. None of these are currently included in the X/OPEN definition. The X/OPEN definition also applies restrictions to the ANS-based parts of the LEVEL II definition.

Whilst the X/OPEN COBOL definition is based on the specification of a particular product, the means of implementation across the systems of the X/OPEN members may vary. Any particular system may support extensions beyond the facilities identified, but their use is likely to impede portability.

The X/OPEN COBOL definition is given in detail in "COBOL LANGUAGE" and its relationship to the 1974 standard is clearly shown.

The definition is given in terms of the command syntax derived from the LEVEL II COBOL Reference Manual. The semantics of the ACCEPT and DISPLAY verbs are defined in "COBOL LANGUAGE". The semantics of all other elements of the language are defined by the "X3.23-1974" standard.

5.3 FORTRAN

The X/OPEN definition for FORTRAN is the formal definition given in the American National Standards document "FORTRAN 77, ANS X3.9 - 1978". This has had wide-scale acceptance throughout the world and there are many certified compilers available.

The majority of FORTRAN compilers, while adhering to the basic FORTRAN 77 standard, also offer extensions beyond that standard. There is little compatibility in these extensions between compilers and they do not form part of the X/OPEN definition. Developers are warned that use of these extensions will affect the portability of FORTRAN programs.

5.4 PASCAL

The current X/OPEN definition for PASCAL is the formal definition given in the International Organisation for Standardisation document "Programming Languages - PASCAL " ISO 7185-1983 (level 1).

This is well accepted throughout the world and there are significant numbers of certified compilers available.

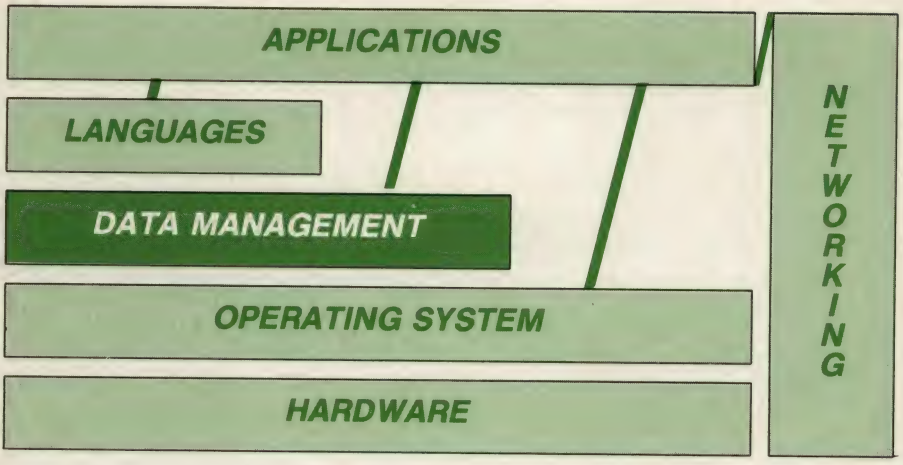
In order to enchance the portability of programs among PASCAL implementations on X/OPEN systems, the X/OPEN Group has decided to give a uniform definition for certain features designated as implementation-defined in ISO 7185.

- When the required identifiers "input" and "output" occur as program parameters they shall be bound by default to the external system files STDIN and STDOUT respectively.
- There shall be no implementation dependent restrictions on the base-type of a set-type which disallow 0 as the minimal ordinal value and 255 as the maximum ordinal value of that base-type.
- (lazy input) the underlying data transfer action required for a call to the predefined procedure "get" for a textfile (both explicit and where implied by "reset" and "read"), shall be postponed until one of the following events, if any, whichever occurs first :
  a. access to the buffer-variable of the file
  b. the buffer-variable of the file is passed as an actual variable parameter to a procedure or function
  c. a call to the predefined function "eoln" for that file
  d. a call to the predefined function "eof" for that file

The majority of Pascal compilers also offer extensions beyond the current ISO Standard PASCAL definition. There is little compatibility in these extensions between compilers and developers are warned that use of these extensions may affect the portability of PASCAL programs.

Chapter 6

Data Management



6.1 INTRODUCTION

The input/output facilities supported by System V consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data Management is a key element in the integration of applications. Applications written in a variety of languages must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

Addressing these issues, the X/OPEN Group defines interfaces for the creation, management and manipulation of indexed files, generally known as the Indexed Sequential Access Method (ISAM) and for access to relational database management systems, the standard Relational Database Language (SQL).

The availability of these interfaces on X/OPEN systems will not only provide application portability, but will ease and encourage integration.

6.2 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

The X/OPEN definition for ISAM, which is contained in "INDEXED SEQUENTIAL ACCESS METHOD", is a major subset of the specification of the C-ISAM product, Version 2.10, published by Informix Corporation

The full specification of C-ISAM contains implementation details specific to that product, in addition to the definition of the interface available to applications. Only the applications interface forms part of the X/OPEN definition; implementation details specific to C-ISAM have been omitted. Indeed, there are alternative implementations available on particular member systems.

6.3 RELATIONAL DATABASE LANGUAGE (SQL)

To reflect the growing significance of Relational Data Base systems, the X/OPEN group has defined application interfaces embedded within high level "host" languages to a relational database management system for a free-standing database.

The widely accepted standard for access to relational data base is that defined in the American National Standard document Relational Database Language (SQL) "ANS X3.135-1986".

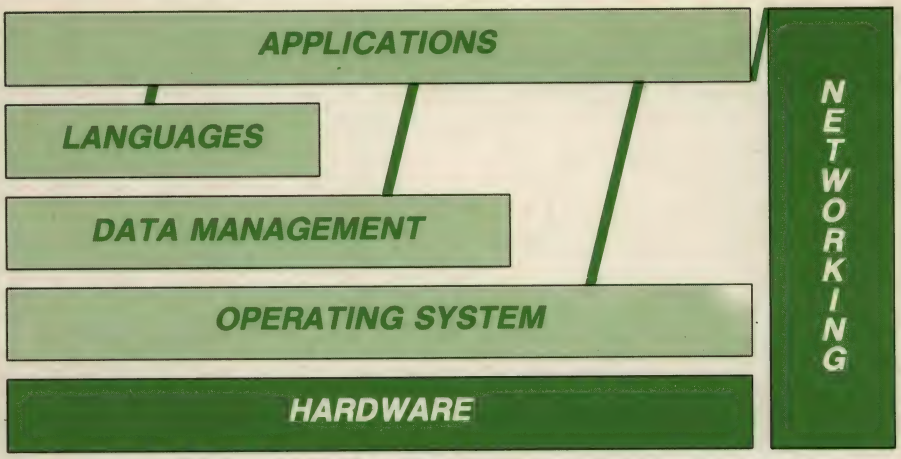
The X/OPEN definition is based closely on "X3.135-1986" but taking careful account of the capabilities of the leading relational database management systems currently available. The X/OPEN group has worked closely with the vendors of these products throughout.

"X3.135-1986" allows for two levels of compliance, Level 1 and Level 2. Most existing products comply only at Level 1 although it is expected that a significant number of products will have achieved full compliance with Level 2 before the end of 1987. "X3.135-1986" Level 1 SQL is not an adequate definition for application developers, since it leaves too many areas as implementor-defined. In preparing its definition, the X/OPEN group has examined these areas carefully and an agreed X/OPEN approach defined.

The X/OPEN SQL definition is contained in "RELATIONAL DATABASE LANGUAGE (SQL)", and contains a full description of the syntax and semantics of SQL together with a detailed comparison between the X/OPEN definition and the "ANS X3.135-1986" standard.

Chapter 7

Source Code Transfer Between Machines



7.1 INTRODUCTION

One of the major problems inhibiting the porting of applications between UNIX system derivatives is that of incompatible media standards and the physical problems of transferring source code in machine readable form.

The X/OPEN Group takes this problem seriously and has agreed common standards for the transfer of source code. Detailed standards are defined in "SOURCE CODE TRANSFER".

Standards are defined for transfer of 5¼" floppy discs and ½" magnetic tape between machines. Because of the different nature of X/OPEN systems, ranging from single user work stations to large mainframes, it is not possible to define formats which are portable across the whole range. Defining standards for both floppy discs and ½" magnetic tape gives the highest practical coverage of systems.

Current differences in the physical recording formats between cartridge tape devices prevents the definition of a standard for this popular medium.

Because of restrictions imposed by existing hardware, some X/OPEN members are not able to support the floppy disc standard.

7.2 FLOPPY DISC STANDARD

As exchange media, the X/OPEN group defines standards for 40 and 80 track floppy discs. It is intended that the prime format should be 80 track, with 40 track retained for compatibility with personal computers. X/OPEN systems equipped only with 80 track disc drives will offer the facility to read 40 track floppy discs by skipping alternate tracks.

7.3 MAGNETIC TAPE

The X/OPEN standard for magnetic tape covers ½" magnetic tape, with a number of different recording formats and densities. The prime format is 9 track Phase Encoded at 1600 bits per inch.

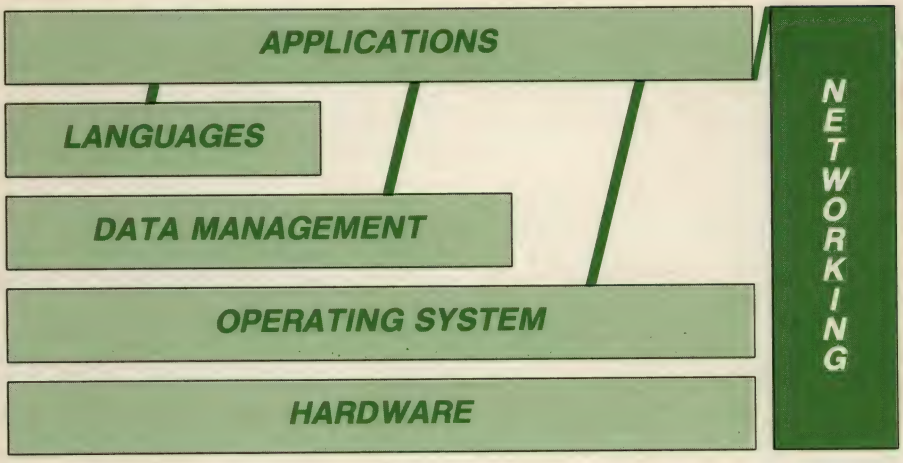
7.4 UTILITIES

"SOURCE CODE TRANSFER" includes the definition of two alternative utilities for the archiving of files to the transfer medium and their subsequent retrieval, *tar* and *cpio*.

In addition, guidelines are given on the use of direct machine to machine connection and the *uucp* utility, as a means of transferring files between X/OPEN systems.

Chapter 8

Networking and Communications



8.1 NETWORKING AND COMMUNICATION

The general target for computer data communications is interworking between systems of different types from different suppliers. Future X/OPEN definitions for such open systems interworking can be expected to embrace the ISO OSI standard.

Two services specific to systems supporting the System V Interface have been identified. These are "Generalised Inter Process Communication" (IPC) and "Distributed File System".

Many current commercial applications are supported via interactive transaction processing systems. X/OPEN definitions in this area can be expected to be in line with emerging International Standards.

8.2 OPEN SYSTEMS INTERCONNECTION

For interworking to be possible, systems must have common methods of describing both tasks and data, and must be capable of functioning in a defined manner. To describe interconnection, an architectural model is required. The International Organisation for Standardisation, ISO, has developed the "Reference Model for Open Systems Interconnection" (IS7498). This is often referred to as the "ISO OSI model" or the "ISO 7-layer model". This model sets a framework into which protocol and service standards can be set.

The ISO OSI target is to have a complete set of protocol and service definitions which comply with the 7-layer model and which are internationally agreed and published as ISO standards.

A complete set of ISO OSI standards does not yet exist. Where standards are available, they contain options. For practical systems to be built, it is necessary to have a very clear definition of standards to be adopted and the options to be used.

To provide a clear statement of the standards to be used, a specialist working group has been formed by the 12 major European vendors who propose the technical objectives for "ESPRIT", the "European Strategic Programme for Research into Information Technology". This is called the "Standards Promotion and Application Group, SPAG".

Where ISO standards do not yet exist, interim standards from one of the national or international standards bodies are adopted by SPAG. Such standards are expected to form the basis of future ISO standards. SPAG is not itself a standard making body. Its recommendations will reflect evolving standards.

X/OPEN member companies are committed to the ISO OSI target and the adoption of ISO standards. The group will monitor SPAG recommendations.

X/OPEN intends to define application interfaces for access to OSI services to ensure the portability of applications and library routines.

8.3 GENERALISED INTER-PROCESS COMMUNICATION, IPC

UNIX operating systems provide limited IPC capabilities in the form of "pipes" and "fifos". Kernel extensions within the AT&T System V Interface Definition provide some further IPC mechanisms for the passing of messages between processes in the same memory address space. These extensions were omitted from issue 1 of the X/OPEN definition because it was believed that a much more generalised mechanism for peer to peer communication between processes, either in the same physical machine or in different machines connected via some communications medium is needed.

The X/OPEN group is working on the definition of such a mechanism and but in the short term, it is recognised that there are some applications which need access to such IPC capabilities as currently exist. "XVS INTER-PROCESS COMMUNICATION" gives detailed definitions for

- Message passing between processes.
- Shared memory.
- Semaphores.

It must be recognised that these routines cannot be implemented on all hardware architectures and hence are optional in the X/OPEN definition. However, where the interfaces are supported, the behaviour will be as defined.

8.4 DISTRIBUTED FILE SYSTEM

There is an increasing requirement to be able to access data contained within UNIX File Systems on machines connected together by a local area network from any system on that network. The totality of data accessible in this way can be regarded as a "distributed file system".

The X/OPEN Group regards the way in which local resources are made available to other systems to be a matter of system administration, and does not intend to publish detailed definitions.

The only aspect of such systems which is relevant to the application developer is the behaviour of the system towards applications at run-time.

The characteristics of any distributed file system supported by X/OPEN systems are:

- Access to the distributed file system is via the standard input/output system calls, and is identical for local and remote files.
- No changes are necessary to existing applications. Binary copies of existing applications are able to access a distributed file system, subject to the requirement that the data within a file is in a compatible format.
- Naming of remote items follows the same syntax as for local items and no new naming conventions are required.
- File locking applies across the network.

8.5 DISTRIBUTED TRANSACTION PROCESSING

Many commercial applications require interactive transaction processing facilities and the X/OPEN Group considers the provision of such facilities to be of key importance.

International Standards Organisations have not made the expected progress in this area.

The X/OPEN Group intends to take action to improve this situation.

X/OPEN

PORTABILITY GUIDE

THE X/OPEN C LANGUAGE DEFINITION

Contents

Chapter 1

Introduction

Together, the C-language and the System V interface provide the foundation for application portability. "XVS SYSTEM CALLS AND LIBRARIES" and "XVS COMMANDS AND UTILITIES" define the X/OPEN System V Interface. This part covers the C language and gives guidelines for portability when writing C code.

The ANS committee, X3J11, is currently working towards a standard for the C programming language. X/OPEN is represented on that committee by member companies and intends to adopt the standard once it is established as a practical reality. Where reference is made to that standard, it should be taken to mean the latest available draft (see also Section 3.14).

In the meantime, X/OPEN adopts the definition of the C-language given in Chapter 2 of the AT&T "UNIX™ System V Programming Guide, Release 2.0". This definition is reproduced in Chapter 2. Machine-specific information has been removed, and some minor editorial alterations have been made. This document is *not* a tutorial and assumes familiarity with the language on the part of the reader.

The C language, as defined, does not guarantee portability. It is possible, using valid C constructs, to write programs that are machine-specific. In Chapter 3, advice is given on writing portable application programs in C. It is anticipated that this chapter will grow in future editions, and readers are invited to contribute additional material.

It is a declared intention of X/OPEN to develop a co-ordinated and integrated approach to internationalisation. Towards this end, Chapter 3 also gives advice to application writers on structuring C-language programs that are intended to operate in more than one natural language / character-set environment.

A draft of the ANS X3J11 standard has been published, and this indicates the need for portability guidelines to ensure that programs will compile correctly when using future compilers that support the standard. This issue is also addressed in Chapter 3.

The description of the "C Program Checker - *lint*" featured in the AT&T "UNIX System V Programming Guide, Release 2.0", is reproduced in Chapter 4. Use of this program is strongly recommended since it enforces a number of portability restrictions, in addition to carrying out general checks on a C program.

Chapter 2

C Language Definition

2.1 LEXICAL CONVENTIONS

There are six classes of tokens: identifiers; keywords; constants; strings; operators; and other separators. Blanks, tabs, new-lines, and comments (collectively "white space") as described below are ignored except that they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1.1 Comments

The characters `/*` introduce a comment which terminates with the characters `*/`. Comments do not nest.

2.1.2 Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Upper case and lower case letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names.

2.1.3 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**extern**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran** and **asm**.

2.1.4 Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "NAMES". Hardware characteristics that affect sizes are summarised in "Hardware Characteristics" under "LEXICAL CONVENTIONS".

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7**. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values **10** through **15**. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a **long** constant. As discussed below, on some machines integer and **long** values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in **'x'**. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is **0** (not followed by a digit), which indicates the character **NUL**. If the character following a backslash is not one of those specified, the behaviour is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Enumeration Declarations" under "DECLARATIONS") have type **int**.

2.1.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialised with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

2.1.6 Hardware Characteristics

The following table summarises certain hardware properties that vary from machine to machine.

It is the responsibility of the programmer to check that the sizes of data structures on any particular machine are sufficient for requirements.

Type	Typical 16-bit Processor (ASCII)	Typical 32-bit Processor (ASCII)
char	8 bits	8 bits
int	16	32
short	16	16
long	32	32
float	32	32
double	64	64
float range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$

2.2 SYNTAX NOTATION

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt", so that:

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarised in "SYNTAX SUMMARY".

2.3 NAMES

The C language bases the interpretation of an identifier upon two attributes of the identifier - its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

2.3.1 Storage Class

There are four declarable storage classes:

- Automatic
- Static
- External
- Register.

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon re-entry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

2.3.2 Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, **char** may be signed or unsigned by default.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

The properties of **enum** types (see "Enumeration Declarations" under "DECLARATIONS") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation.

Single-precision floating point (**float**) and double-precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. **Char**, **int** of all sizes whether **unsigned** or

not, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *Arrays* of objects of most types
- *Functions* which return objects of a given type
- *Pointers* to objects of a given type
- *Structures* containing a sequence of objects of various types
- *Unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

2.4 OBJECTS AND LVALUES

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

2.5 CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarised under "Arithmetic Conversions". The summary will be supplemented as required by the discussion of each operator.

2.5.1 Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. On machines which do sign-extend, **char** variables range in value from -128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, if a **char** is 8 bits, \377 has the value **-1**.

When a longer integer is converted to a shorter integer or to a **char**, it is truncated on the left. Excess bits are simply discarded.

2.5.2 Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a **float**.

2.5.3 Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

2.5.4 Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of "Additive Operators". Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

2.5.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to **unsigned** and the result is **unsigned**. The value is the least unsigned integer congruent to the signed integer (modulo $2^{word\text{-}size}$). In a 2's complement representation, this conversion is conceptual; there is no actual change in the bit pattern.

When an unsigned short integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

2.5.6 Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions".

1. Any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.
2. If either operand is **double**, the other is converted to **double** and that is the type of the result.
3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.
4. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.
5. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
6. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
7. Otherwise, both operands must be **int**, and that is the type of the result.

2.5.7 Void

The (non-existent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a non-existent value, such an expression may be used only as an expression statement (see "Expression Statement" under "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" under "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

2.6 EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarised by the grammar in "SYNTAX SUMMARY".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

2.6.1 Primary Expressions

Primary expressions involving ., —>, subscripting, and function calls group left to right.

primary-expression:
- *identifier*
- *constant*
- *string*
- (*expression*)
- *primary-expression* [*expression*]
- *primary-expression* ($expression\text{-}list_{opt}$)
- *primary-expression* . *identifier*
- *primary-expression* —> *identifier*

expression-list:
- *expression*
- *expression-list* , *expression*

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string. (There is an exception in certain initialisers; see "Initialisation" under "DECLARATIONS").

A parenthesised expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type of the result is "...". The expression **E1[E2]** is identical (by definition) to ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, * and +, respectively. The implications are summarised under "Arrays, Pointers, and Subscripting" under "TYPES REVISITED".

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "DECLARATIONS".

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from — and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1—>MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure and Union Declarations" under "DECLARATIONS".

2.6.2 Unary Operators

Expressions with unary operators group right to left.

unary-expression:
- * *expression*
- **&** *lvalue*
- **-** *expression*
- **!** *expression*
- **~** *expression*
- **++** *lvalue*
- **--** *lvalue*
- *lvalue* **++**
- *lvalue* **--**
- **(** *type-name* **)** *expression*
- **sizeof** *expression*
- **sizeof (** *type-name* **)**

The unary * operator means *indirection*; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary **&** operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary **-** operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary **+** operator.

The result of the logical negation operator **!** is one if the value of its operand is zero, or zero if the value of its operand is non-zero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The **~** operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix **++** is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x=x+1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesised name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in "Type Names" under "DECLARATIONS".

The **sizeof** operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesised type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type***)** is taken to be a unit, so the expression **sizeof(***type***)-2** is the same as **(sizeof(***type***))-2**.

2.6.3 Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
expression * *expression*
expression / *expression*
expression % *expression*

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b + a%b** is equal to **a** (if **b** is not 0).

2.6.4 Additive Operators

The additive operators + and - group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression - *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

2.6.5 Shift Operators

The shift operators << and >> group left to right.

First, both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:
expression << *expression*
expression >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions.

The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

2.6.6 Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

2.6.7 Equality Operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be NULL.

2.6.8 Bitwise AND Operator

and-expression:
expression **&** *expression*

The **&** operator is associative, and expressions involving **&** may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

2.6.9 Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

2.6.10 Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | *expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

2.6.11 Logical AND Operator

logical-and-expression:
expression **&&** *expression*

The **&&** operator groups left to right. It returns 1 if both its operands evaluate to non-zero; 0 otherwise. Unlike **&**, **&&** guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

2.6.12 Logical OR Operator

logical-or-expression:
expression || *expression*

The || operator groups left to right. It returns 1 if either of its operands evaluates to non-zero; 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

2.6.13 Conditional Operator

conditional-expression:
expression **?** *expression* : *expression*

Conditional expressions group right to left. The first expression is evaluated; and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both are pointers of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

2.6.14 Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are

separate tokens.

assignment-expression:
lvalue = *expression*
lvalue += *expression*
lvalue -= *expression*
lvalue *= *expression*
lvalue /= *expression*
lvalue %= *expression*
lvalue >>= *expression*
lvalue <<= *expression*
lvalue &= *expression*
lvalue ^= *expression*
lvalue |= *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Otherwise, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behaviour of an expression of the form **E1** *op*= **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* (**E2**); however, **E1** is evaluated only once. In += and -=, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "Additive Operators". All right operands and all nonpointer left operands must have arithmetic type.

2.6.15 Comma Operator

comma-expression:
expression , *expression*

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initialisers (see "Initialisation" under "DECLARATIONS"), the comma operator as described in this subpart can only appear in parentheses.
For example,

```
f(a, (t = 3, t + 2), c)
```

has three arguments, the second of which has the value 5.

2.7 DECLARATIONS

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the *declarator-list* contain the identifiers being declared. The *decl-specifiers* consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

2.7.1 Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "Typedef" for more information. The meanings of the various storage classes were discussed in "NAMES".

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one *sc-specifier* may be given in a declaration. If the *sc-specifier* is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Functions are *never* automatic.

2.7.2 Type Specifiers

The type-specifiers are

type-specifier:
    basic-type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier
basic-type-specifier:
    basic-type
    basic-type basic-type-specifier
basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most one *type-specifier* may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the *type-specifier* is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations". Declarations with **typedef** names are discussed in "Typedef".

2.7.3 Declarators

The *declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initialiser.

declarator-list:
    init-declarator
    init-declarator , *declarator-list*

init-declarator:
    declarator initialiser$_{opt}$

Initialisers are discussed in "Initialisation". The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer.

Declarators have the syntax:

declarator:
identifier
(*declarator*)
* *declarator*
declarator ( )
declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

2.7.4 Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a *type-specifier* (like **int**, etc.) and **D1** is a *declarator*. Suppose this declaration makes the identifier have type "... **T**", where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

***D**

the type of the contained identifier is "... pointer to **T**".

If **D1** has the form

D( )

then the contained identifier has the type "... function returning **T**".

If **D1** has the form

D[*constant-expression***]**

or

D[ ]

then the contained identifier has type "... array of **T**". In the first case, the constant expression is an expression whose value is determinable at compile time, whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "CONSTANT EXPRESSIONS"). When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which

specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialisation. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result yields an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

2.7.5 Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { *struct-decl-list* }
struct-or-union identifier { *struct-decl-list* }
struct-or-union identifier

struct-or-union:
struct
union

The *struct-decl-list* is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , *struct-declarator-list*

In the usual case, a *struct-declarator* is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:
declarator
declarator : *constant-expression*
: *constant-expression*

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on some machines, left to right on others.

A *struct-declarator* with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependent boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be **unsigned**, on some systems. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator **&** may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier* { *struct-decl-list* }
union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or *union tag*) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

struct tnode s, *sp;

declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort. With these declarations, the expression

sp—>count

refers to the **count** field of the structure to which *sp* points;

s.left

refers to the left subtree pointer of the structure *s*; and

s.right—>tword[0]

refers to the first character of the **tword** member of the right subtree of **s**.

2.7.6 Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The identifiers in an *enum-list* are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the *enum-specifier* is entirely analogous to that of the structure tag in a *struct-specifier*; it names a particular enumeration.

For example,

```
enum colour { chartreuse, burgundy, claret=20, winedark };
...
enum colour *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **colour** the enumeration-tag of a type describing various colours, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

2.7.7 Initialisation

A declarator may specify an initial value for the identifier being declared. The initialiser is preceded by = and consists of an expression or a list of values nested in braces.

initialiser:
= *expression*
= { *initialiser-list* }
= { *initialiser-list* , }

initialiser-list:
expression
initialiser-list , *initialiser-list*
{ *initialiser-list* }
{ *initialiser-list* , }

All the expressions in an initialiser for a static or external variable must be constant expressions, which are described in "CONSTANT EXPRESSIONS", or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialised by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialised are guaranteed to start off as zero. Automatic and register variables that are not initialised will start with unpredictable values.

When an initialiser applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initialiser consists of a brace-enclosed, comma-separated list of initialisers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initialisers in the list than there are members of the aggregate, then the aggregate is padded with zeros.

It is not permitted to initialise unions or automatic aggregates.

Braces may in some cases be omitted. If the initialiser begins with a left brace, then the succeeding comma-separated list of initialisers initialises the members of the aggregate; it is erroneous for there to be more initialisers than members. If, however, the initialiser does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialise the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialised by a string. In this case successive characters of the string initialise the members of the array.

For example,

```
int x[ ] = { 1, 3, 5 };
```

declares and initialises **x** as a one-dimensional array which has three members, since no size was specified and there are three initialisers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialisation: **1**,**3**, and**5** initialise the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialise **y[1]** and **y[2]**. The initialiser ends early and therefore **y[3]** is initialised with **0**. Precisely the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initialiser for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initialises the first column of **y** (regarded as a two-dimensional array) and leaves the rest **0**.

Finally,

```
char msg[ ] = "Syntax error on line %s \n";
```

shows a character array whose members are initialised with a string.

2.7.8 Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(*abstract-declarator*)
* *abstract-declarator*
abstract-declarator ( )
abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(*abstract-declarator*)

the *abstract-declarator* is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the *abstract-declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *( )
int (*)( )
int (*[3])( )
```

name respectively the types "integer", "pointer to integer", "array of three pointers to integers", "pointer to an array of three integers", "function returning pointer to integer", "pointer to function returning an integer", and "array of three pointers to functions returning an integer".

2.7.9 Typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
identifier

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators". For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int** ", and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

2.8 STATEMENTS

Except as indicated, statements are executed in sequence.

2.8.1 Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

2.8.2 Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initialisations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initialisations are not performed. Initialisations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialisation is not permitted.

2.8.3 Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; and if it is non-zero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The **else** ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

2.8.4 While Statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

2.8.5 Do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

2.8.6 For Statement

The **for** statement has the form:

for ($exp\text{-}1_{opt}$; $exp\text{-}2_{opt}$; $exp\text{-}3_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialisation for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an increment that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

2.8.7 Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in "CONSTANT EXPRESSIONS".

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default** prefix, control passes to the prefixed statement. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "Break Statement".

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initialisations of automatic or register variables are ineffective.

2.8.8 Break Statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

2.8.9 Continue Statement

The statement

continue ;

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is, to the end of the loop. More precisely, in each of the statements

```
while (...)     do              for (...)
{               {               {
    ...             ...             ...
contin: ;       contin: ;       contin: ;
}               } while (...);  }
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, see "Null Statement".)

2.8.10 Return Statement

A function returns to its caller by means of the **return** statement which has one of the forms

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesised.

2.8.11 Goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The *identifier* must be a label (see "Labeled Statement") located in the current function.

2.8.12 Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the *identifier* as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See "SCOPE RULES".

2.8.13 Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

2.9 EXTERNAL DEFINITIONS

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "DECLARATIONS") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

2.9.1 External Function Definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only *sc-specifiers* allowed among the *decl-specifiers* are **extern** or **static**; see "Scope of Externals" in "SCOPE RULES" for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

function-declarator:
declarator **(** *parameter-list*$_{opt}$ **)**

parameter-list:
identifier
identifier **,** *parameter-list*

The function-body has the form

function-body:
declaration-list$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here **int** is the *type-specifier*; **max(a, b, c)** is the *function-declarator*; **int a, b, c;** is the *declaration-list* for the formal parameters; { ... } is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of . . . " are adjusted to read "pointer to . . . ."

2.9.2 External Data Definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be **extern** (which is the default) or **static** but not **auto** or **register**.

2.10 SCOPE RULES

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers which is characterised by the rule that references to the same external identifier are references to the same object.

2.10.1 Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "DECLARATIONS") that tags, identifiers associated with ordinary variables and labels, and identifiers associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
    auto int distance;
    ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

2.10.2 Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the

type and size specified in the definition are compatible with those specified by each function which references the data.

It is illegal to explicitly initialise any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identifiers without initialisation indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a multi-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

2.11 COMPILER CONTROL LINES

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

2.11.1 Token Replacement

A compiler-control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier*(*identifier*,...) *token-string*$_{opt}$

where there is no space between the first identifier and the (, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is re-scanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants", as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#define**d identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

2.11.2 File Inclusion

A compiler control line of the form

#include *"filename"*

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include <*filename*>

searches only the specified or standard places and not in the directory of the file containing the **#include**. (How the places are specified is not part of the language).

#includes may be nested.

2.11.3 Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the *restricted-constant-expression* evaluates to non-zero. (Constant expressions are discussed in "CONSTANT EXPRESSIONS"; the following additional restrictions apply here: the constant expression may not contain **sizeof**, casts, or an enumeration constant.)

A restricted constant expression may also contain the additional unary expression

defined *identifier*
or
defined(*identifier* **)**

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in *restricted-constant-expressions* are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#if defined(***identifier***)**. A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if !defined(***identifier***)**.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

2.11.4 Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

#line *constant "filename"*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by *"filename"*. If *"filename"* is absent, the remembered file name does not change.

2.12 IMPLICIT DECLARATIONS

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ...", it is implicitly declared to be **extern**.

In an expression, an identifier followed by **(** and not already declared is contextually declared to be "function returning **int**".

2.13 TYPES REVISITED

This part summarises the operations which can be performed on objects of certain types.

2.13.1 Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the —> or the . must specify a member of the aggregate pointed to or named by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int     type;
        } n;
        struct
        {
                int     type;
                int     intnode;
        } ni;
        struct
        {
                int     type;
                float   floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

2.13.2 Functions

There are only two things that can be done with a function: call it or take its address. If the name of a function appears in an expression not in the function-

name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

2.13.3 Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator **[ ]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules which apply to **+**, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2**th member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If **E** is an *n*-dimensional array of rank i×j×...×k, then **E** appearing in an expression is converted to a pointer to an (*n-1*)-dimensional array with rank j×...×k. If the ***** operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (*n-1*)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

2.13.4 Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under "EXPRESSIONS" and "Type Names" under "DECLARATIONS".

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc( );
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the *use* of the function is portable.

2.14 CONSTANT EXPRESSIONS

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initialisers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

+ - * / % & | ^ << >> == != < > <= >= && ||

or by the unary operators

- ~

or by the ternary operator

?:

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initialisers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary **&** operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary **&** can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initialisers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

2.15 SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

2.15.1 Expressions

The basic expressions are:

expression:
primary
** expression*
& lvalue
- expression
! expression
~ expression
++ lvalue
-- lvalue
lvalue ++
lvalue --
sizeof *expression*
sizeof (*type-name***)**
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . identifier
primary —> identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary —> identifier
** expression*
(lvalue)

The primary-expression operators

() [] . —>

have highest priority and group left to right. The unary operators

* & - ! ~ ++ -- **sizeof**(*type-name*)

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:

* / %
\+ -
>> <<
< > <= >=
== !=
&
^
|
&&
||

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:

= += -= *= /= %= >>= <<= &= ^= |=

The comma operator has the lowest priority and groups left to right.

2.15.2 Declarations

declaration:
decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier
basic-type-specifier:
basic-type
basic-type basic-type-specifier
basic-type:
char
short
int
long
unsigned
float
double
void

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , enumerator

enumerator:
identifier
identifier = constant-expression

init-declarator-list:
init-declarator
init-declarator , init-declarator-list

init-declarator:
declarator initialiser$_{opt}$

declarator:
identifier
(declarator)
** declarator*
declarator ( )
declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
struct { *struct-decl-list* }
struct *identifier* { *struct-decl-list* }
struct *identifier*
union { *struct-decl-list* }
union *identifier* { *struct-decl-list* }
union *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initialiser:
= expression
= { initialiser-list }
= { initialiser-list , }

initialiser-list:
expression
initialiser-list , initialiser-list
{ initialiser-list }
{ initialiser-list , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ( )
abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
identifier

2.15.3 Statements

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for (*exp*$_{opt}$; *exp*$_{opt}$; *exp*$_{opt}$) *statement*
switch (*expression*) *statement*
case *constant-expression* : *statement*
default : *statement*
break ;
continue ;
return ;
return *expression* ;
goto *identifier* ;
identifier : *statement*
;

2.15.4 External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
extern *declaration ;*
static *declaration ;*

2.15.5 Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier(identifier,...) token-string*$_{opt}$
#undef *identifier*
#include *"filename"*
#include *<filename>*
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant "filename"*$_{opt}$

Chapter 3

Portability

3.1 GENERAL

The portability areas addressed below arise from differences in the underlying hardware and from variances that can arise from the actual C compiler in use. It is not practically possible to provide a definitive guide to portability. However, general recommendations and hints can be given, and attention drawn to specific problems and possible solutions.

Writing highly portable C programs demands skill in coding, careful testing (preferably on machines of different types), and knowledge of the types of portability problems that have previously arisen.

3.2 DATA ALIGNMENT

Different processors align data items differently. For example, some microprocessors require data types longer than one byte to be aligned on word boundaries, whereas others do not. Therefore any data that lies in structures or unions may be aligned on different boundaries.

A portable program should not make any assumptions regarding a particular alignment.

It is common practice in C programs to write structures to files as a "record", the reason being that such records can then be read back into the same type of structure, and its elements accessed accordingly. However, it is important to remember that this operation is only safe if the records have been written by a program running on the same type of processor, *and* compiled using a compatible compiler.

Because the sizes, alignments and byte-order of the objects inside a structure are determined by the processor architecture and the compiler, such binary record-structured files are not portable.

3.3 BIT AND BYTE ORDERING

The sequencing order of bytes in memory varies from machine to machine, as does the ordering of bits in bit fields. Portable programs should not make any assumptions regarding the layout of bits, bytes or words in memory. In particular, multi-character constants should not be used, since the memory layout is not defined.

3.4 VARIABLE NAMES

Different compilers may allow different maximum lengths for the names of variables. For portability, assume the maximum to be eight characters. However, as some linkers apply a limit below this, six characters is the advised maximum for external

names, and these should be unique irrespective as to whether they are in upper or lower case. Care should be taken not to misspell variable names after the eighth (sixth for external names) character since these will be taken as unique variables by a compiler which accepts a longer name length. For further explanation, refer to Chapter 2 for the definition of the permitted lengths of variable names.

3.5 TRANSLATION LIMITS

There are many translation limits that may differ between implementations of C. Fortunately, most of these will be discovered at translation time. Examples are the maximum length of a string and the maximum size in bytes of an object. The ANS X3J11 draft standard covers many of these limits, and should be taken as guidance on reasonable limits to expect from implementations.

3.6 LENGTHS OF DATA TYPES

The data type **int** is usually mapped as the most natural integer type of a particular machine, and therefore can have different maximum and minimum integer values on different machines. On most machines, the data types **short** and **long** occupy two and four bytes respectively. C gives no guarantees about precisional range for **float** or **double** types. For full portability, variables should be declared as appropriate **typedef** types declared in header files. Where required, the actual sizes of types should be determined by use of the **sizeof** operator.

Most C programmers are aware of the problems that can be caused by different word-lengths on different machines. In the context of portability, the major source of difficulties is the assumption that the **int** variables have more than 16 bits - which is not guaranteed. Code written on machines that do offer 32 bit **int** variables often fails to be fully portable to machines with 16 bit **int**s. This is for several reasons:

- More than 16-bit values are needed - an occasional problem.
- Incorrect use of formats in functions such as *printf*(3S) and *scanf*(3S), where the same format - eg., *%d* - has been used for both **long** and **int** arguments. This only works if both types have the same number of bits, and fails when the code is tried elsewhere (*%d* is for **int**, *%ld* is for **long**).
- Often, passing of incorrect types to library routines can also cause problems. For example, the following incorrect *lseek*(2) library call will work on most 32-bit machines. On most 16-bit machines, it will not.

```
int fdes, offset;
lseek( fdes, offset, 0 );
```

The correct version would define *offset* to be of type **long**. Fortunately, the *lint* program will identify and warn about any problems of this type.

- The types of constants should be made explicit, especially long constants, e.g. *0L*. A common problem is porting software from a 32-bit to a 16-bit machine which has the following code:

```
int fdes;
lseek(fdes, 0, 0); /* Non-portable */
                   /* second parameter should be 0L */
```

it should be written as

```
int fdes;
lseek(fdes, 0L, 0);          /* Portable */
```

Implicit long constants can also cause problems. On a machine with 32-bit integers

```
f(40000);
```

passes an **int** argument, but on a typical 16-bit machine the constant *40000* would be **long**.

- The assignment of **long** to **int** may cause a loss in precision and/or the loss of the sign information. The *lint* program will draw attention to any problems of this type.
- Bit masks should be made data length independent, e.g., to unset the lowest bit in *xyz* use *(xyz & (~1))*.

3.7 MISUSE OF POINTERS

It is a serious error to assume that pointer-to-one-type bears much resemblance to pointer-to-some-other-type. On many machines, pointers and **int** or **long** variables are the same length, and compilers should warn when pointers are assigned to such variables, or vice versa. However, in an attempt to re-use certain variables, some programs assume that this can be done safely and the compiler warnings can often be suppressed through the use of casts.

Be careful not to do such things, unless explicitly performing a non-portable task such as the examination of known storage locations.

Furthermore, it is not even safe to assume that pointers of different types can be intermixed. The only guarantee given by C is that a pointer can be cast to **(char *)** and back.

The formal language definition states that the integer constant *0* is the only non-pointer value that can be assigned to, or compared with, the value of any pointer. This is the value of the NULL pointer but, as is indicated below, it is recommended that explicit use of 0 should not be made.

Some machines are unable to support C's assertion that the value *0* is the value of a NULL pointer, although they adhere to the other semantics of the language. For portability to these machines as well, it is recommended that instead of using *0*, all programs should use the NULL constant declared in <**stdio.h**>. For example:

```
#include <stdio.h>
int *p;

p = NULL;
if(p != NULL){
    fprintf(stderr, "Pointer problem \n");
    abort();
}
```

Another common, mistaken practice is to de-reference such a NULL pointer. Some programs assume that, not only does a NULL pointer generally contain *0*, but also the place that it points to contains *0*. There is nothing to justify such an assumption. If it is possible for a pointer to be NULL, then a program must check that it is not before performing any indirection using the pointer. Checking the validity of pointers is particularly recommended in application libraries.

3.8 EVALUATION ORDER AND SIDE-EFFECTS

The evaluation order of function arguments, and the order of side-effects, is implementation-dependent. Portable programs should not make any assumptions with regard to ordering.

Specifically, the use of the ++, -- and assignment (+=, -=, etc.) operators should be avoided if an argument list or expression mentions the object of these operators more than once.

3.9 ARITHMETIC

In a number of cases the behaviour of programs is undefined. The most common example is arithmetic overflow/underflow and shifting by more places than there are bits in a word. Careful program design is essential if this is to be prevented.

The combination of signed and unsigned numbers presents a subtle portability problem. Whenever signed and unsigned numbers are mixed in C, most implementations apply unsignedness preserving rules, i.e., that the signed quantity is converted to unsigned. If the signed variable contains a negative number, it becomes positive when converted. However, some implementations apply value preserving rules, and this is being followed by ANS X3J11, see Section 3.14.

Under the unsignedness preserving rules, if "c" is of type **char** in the expression "**c += CONST;**", the type of *CONST* should be viewed with suspicion. If it is **int** and negative in value, then the correct result will be obtained if this particular machine used signed **char** variables. If they are unsigned, then *CONST* will first become some unexpected positive value, with surprising results.

Fortunately, the *lint* program checker will warn of this problem if it occurs.

A final point on arithmetic. Never use bit-wise operators to look at the high-order bit in a word in order to find out if it is negative. There is no guarantee that 2's complement arithmetic is in use. Indeed, some machines do not use the high-order bits in a word for arithmetic purposes at all.

3.10 EXTERNAL DECLARATIONS

The current definition of C is ambiguous in its description of the effect of the **extern** keyword. This is especially true of **extern** in inner scope, inside a compound statement. For safety, external objects should only be declared at the outermost level. Conflicting declarations involving both **static** and **extern** should not be used either, as in this example.

```
/* conflicting declarations */
extern int ext;
extern static int ext;
```

External objects should be declared at the beginning of every file intending to use them; nowhere else.

3.11 STRUCTURE MEMBERS

A common practice when referring to members of structures is to provide an incompletely qualified name, which is nonetheless unambiguous, since the compiler fills in the missing information. For instance:

```
struct{
        int a;
        struct{
                float f;
                double d;
        }b;
}x;

x.f = 0;
```

The expression **x.f** is not permitted by the Standard. The object **x** has no member called **f**. The Standard requires the full name to be given, so the expression must read **x.b.f**.

3.12 DIRECTIVES

Text used as comment must not be used following **#else** and **#endif**, unless it is surrounded by comment brackets.

Example:

```
/* incorrect */
#ifdef TOKEN

#endif (ifdef TOKEN)


/* permitted */
#ifdef TOKEN

#endif /* (ifdef TOKEN) */
```

3.13 LINT

Since the *lint* checker provides a considerable aid to portable programming, regular use should be made of it during program development. A program cannot begin to be regarded as portable until it has been checked by *lint*. The check should be performed not only on the machine on which the program was developed, but also on every machine to which the program is moved, even though the output of *lint* should be the same on all machines.

3.14 THE ANS X3J11 DRAFT STANDARD

The ANS X3J11 Committee has now released a document, number X3J11/86-151, as the draft proposed American National Standard for the C Programming Language.

X/OPEN, along with AT&T, has expressed a commitment to adopt the Standard when it is a practical reality. However, because the standard has already modified the language in some places, and has more rigorously defined certain aspects of it, it will impact the portability of programs written according to the current definition of C.

The purpose here, therefore, is to identify changes made to the language and to provide guidelines which will help to prevent programs from conflicting with the standard when it is eventually adopted.

A copy of the draft standard can be obtained from:

CBEMA, 311 First Street, NW,
Suite 500, Washington DC 20001, USA.
Tel: 202-737-8888.

The draft was published on 1st October, 1986, and it should be noted that the standard is not yet approved and may be subject to change. Although substantial changes are not anticipated, the recommendations below may be affected by subsequent revisions to the standard.

3.14.1 Keywords

New keywords have appeared in two forms. First, there are new keywords in the language itself. Second, it is now considered to be an error to use external names that are the same as the names found in the Standard Library - unless the Library is not being used.

The new reserved words are:

const
signed
volatile

They must not be used anywhere in a program as identifiers.

The Standard reserves any identifier beginning with an underscore (_) for use as external objects and macros. Such names should not be used as identifiers, even if the Library is not actually being used.

The following list of external and macro identifiers gives the remaining names used by the Standard Library. These identifiers are also reserved if any Library features are being used. Hence, they are only safe for use in a "freestanding" environment, for example an operating system kernel.

abort	abs	acos
asctime	asin	assert
atan	atan2	atexit
atof	atoi	atol
bsearch	BUFSIZ	calloc
ceil	CHAR_BIT	CHAR_MAX
CHAR_MIN	clearerr	CLK_TCK
clock	clock_t	cos
cosh	ctime	DBL_DIG
DBL_EPSILON	DBL_MANT_DIG	DBL_MAX
DBL_MAX_10_EXP	DBL_MAX_EXP	DBL_MIN
DBL_MIN_10_EXP	DBL_MIN_EXP	difftime
div	div_t	EDOM
EOF	ERANGE	errno
exit	exp	fabs
fclose	feof	ferror
fflush	fgetc	fgetpos
fgets	FILE	floor
FLT_DIG	FLT_EPSILON	FLT_MANT_DIG
FLT_MAX	FLT_MAX_10_EXP	FLT_MAX_EXP
FLT_MIN	FLT_MIN_10_EXP	FLT_MIN_EXP
fmod	fopen	fpos_t
fprintf	fputc	fputs

fread
free
freopen
frexp
fscanf
fseek
ftell
fwrite
getc
getchar
getenv
gets
gmtime
HUGE_VAL
INT_MAX
INT_MIN
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
jmp_buf
labs
LDBL_DIG
LDBL_EPSILON
LDBL_MANT_DIG
LDBL_MAX
LDBL_MAX_10_EXP
LDBL_MAX_EXP
LDBL_MIN
LDBL_MIN_10_EXP
LDBL_MIN_EXP
ldexp
ldiv
ldiv_t
localtime
log
log10
longjmp
LONG_MAX
LONG_MIN
L_tmpnam
malloc
memchr
memcmp
memcpy
memmove
memset
mktime
modf
NDEBUG
NULL
offsetof
OPEN_MAX
perror
pow
printf
ptrdiff_t
putc
putchar
puts
qsort
raise
rand
RAND_MAX
realloc
remove
rename
rewind
scanf
SCHAR_MAX
SCHAR_MIN
SEEK_CUR
SEEK_END
SEEK_SET
setbuf
setjmp
setlocale
setvbuf
SHRT_MAX
SHRT_MIN
SIGABRT
SIGFPE
SIGILL
SIGINT
signal
SIGSEGV
SIGTERM
sig_atomic_t
SIG_DFL
SIG_ERR
SIG_IGN
sin
sinh
size_t
sprintf
sqrt
srand
sscanf
stderr
stdin
stdout
strcat
strchr
strcmp
strcoll
strcpy
strcspn
strerror
strftime
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtod
strtok
strtol
strtoul
system
tan
tanh
time

time_t	tm	tmpfile
tmpnam	TMP_MAX	tolower
toupper	UCHAR_MAX	UINT_MAX
ULONG_MAX	ungetc	USHORT_MAX
va_arg	va_end	va_list
va_start	vfprintf	vprintf
vsprintf		

3.14.2 Arithmetic Conversions

The current definition for the arithmetic conversions that are to be followed when different data types are encountered in an expression is known as "unsignedness preserving rules", since if either operand is unsigned then the operation is done unsigned. E.g., if the operands have type **unsigned int** and **long** the operation will be done as **unsigned long**. There is another possible set of rules called value preserving rules, which the ANS X3J11 draft standard and also some current implementations of C use. These differ from the unsignedness preserving rules in that the narrower operand is always converted to have the same type as the wider type, with unsigned types being considered to be wider than their signed equivalent. E.g., if the operands have type **unsigned int** and **long** the operation will be done as **long**.

If there is a chance that changing between the two sets of rules could change the result of an expression then casts should be used to ensure that the required result is obtained.

3.14.3 Characters

The Standard introduces a notation to express characters present in the C alphabet which are not present in ISO646 (invariant subset). The notation uses "tri-graphs", these being the escape sequence **??** followed by one other character. At a very early stage of lexical processing during compilation, the tri-graph sequence is translated into a single character in the C character set.

The only places that this can occur in a well-formed program, according to current definitions of C, is in character constants, strings, and comment. Therefore, to ensure compatibility with the new Standard, it is recommended that the sequence **??** is avoided throughout the source of a C program.

3.14.4 Preprocessor

#define

The operation of the **#define** directive has changed considerably in the Standard. At the simplest level, its use is unchanged, and "normal" use, to define constants and macros, will be relatively unaffected. Specific points to look out for are:

- Replacement of defined names no longer occurs in strings or character constants.

- The name of the macro currently being expanded is not itself expanded if it occurs in the replacement string.
- Recursion and catenation (token pasting) was not portable prior to the Standard. It should not be present in programs in any way.

The Standard defines ways in which replacement inside strings and token catenation can be performed.

3.15 INTERNATIONALISATION AND THE X3J11 DRAFT STANDARD

The X3J11 Draft Proposed Standard for the C Programming Language includes a number of library routines that have been defined to operate internationally; that is, they modify their operation in a manner appropriate to the native language and cultural environment of the user. X/OPEN defines similar facilities in "XVS INTERNATIONALISATION" and there are some areas of overlap.

The X/OPEN definition attempts to respect and maintain compatibility with the X3J11 in two important areas:

- The overall style of the syntax
- Where there is a direct correspondence between the library routines in both systems.

In these areas, it is believed that the X3J11 proposal is unlikely to change as a result of the public review process.

In other areas, the X3J11 proposal has introduced new or alternative routines. It is considered that these are more liable to change, and it would be premature for the X/OPEN definition to adopt them.

Applications writers should, however, take note of the areas of difference between the X/OPEN and X3J11 definitions in order to minimise any later portability problems. A summary of the situation is as follows:

1. General architecture and syntactic style is the same, so program structure should not be impacted.
2. The routines listed below are included in both systems and, within the degree of detail available in the X3J11 proposal at the time of writing, they are believed to be substantially compatible.

isalnum	toupper	atof
isalpha	to lower	strtod
iscntrl		
isdigit	printf	
isgraph	fprintf	
isupper	sprintf	
isprint	scanf	
isspace	fscanf	
ispunct		

3. The X/OPEN enhanced versions of formatted input and output, which allow for variable ordering of parameters, do not have equivalents in the X3J11 proposal.

nl_printf	nl_fprintf	nl_sprintf
nl_scanf	nl_fscanf	nl_sscanf

4. The methods of character ordering and collating are different. X/OPEN has enhanced versions of *strcmp* (*nl_strcmp*) and *strncmp* (*nl_strncmp*). X3J11 introduces the routine *strcoll*, which is used to preprocess data before calling the standard routine.

5. There are different routines for handling time and date. X/OPEN uses an enhanced form of *ctime* (*nl_cxtime*) and *asctime* (*nl_ascxtime*). X3J11 introduces the routine *strftime*.

6. The announcement mechanisms for introducing the user requirements are different.

3.16 INPUT/OUTPUT DEVICES

The handling of I/O devices such as terminals, printers and keyboards is often dependent on the hardware concerned. To insulate applications from changes in I/O devices, code written to handle them should be kept in separate modules which can easily be changed.

The design of applications software should try to avoid depending on a particular form of user interface. Ideally, an application should be capable of working on I/O devices ranging from line by line typewriters to multi-window high resolution graphics devices, simply by changing (or re-configuring) its user interface modules. This is achieved much more easily if it is built in at the design stage, instead of being imposed on an application which already works on one class of I/O devices.

Applications Developers are strongly recommended to use the interfaces described in the part of the Guide titled "XVS TERMINAL INTERFACES", but should still abide by the guidelines on modularity given above. Standards for user interfaces are not yet well established and it is prudent to adopt designs which are capable of easy modification.

Chapter 4

Lint

4.1 GENERAL

The *lint* program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions which nevertheless are legal. The *lint* program accepts multiple input files and library specifications and checks them for consistency.

4.2 USAGE

The *lint* command has the form:

lint [options] files . . . library-descriptors . . .

where *options* are optional flags to control *lint* checking and messages; *files* are the files to be checked which end in **.c** or **.ln**; and *library-descriptors* are the name of libraries to be used in checking the program.

The options that are currently supported by the *lint* command are described under the entry *lint*(1) in the part of the Guide titled "COMMANDS AND UTILITIES".

The names of files that contain C language programs should end with the suffix **.c** which is mandatory for *lint* and the C compiler.

The *lint* program accepts certain arguments, such as:

—ly

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the command:

/*LINTLIBRARY*/

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions.

The *lint* library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The *lint* program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, *lint* checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the **—p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **—n** option can be used to suppress all library checking.

4.3 TYPES OF MESSAGES

The following paragraphs describe the major categories of messages printed by *lint*.

4.3.1 Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The *lint* program prints messages about variables and functions which are defined but not otherwise mentioned. Variables which are declared through explicit **extern** statement but are never referenced, such as the statement

```
extern double sin();
```

will evoke a comment if *sin* is never used. In some cases, these unused external declarations might not be of interest and so can be suppressed by using the —x option with the *lint* command, (note that this agrees with the semantics of the C compiler).

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The —v option is available to suppress the printing of messages about unused arguments. When —v is in effect, no messages are produced about unused arguments except for those arguments unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/*ARGSUSED*/
```

to the program before the function. This has the effect of the —v option for only one function. Also, the comment:

```
/*VARARGS*/
```

can be used to suppress messages about variable numbers of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
/*VARARGS2*/
```

will cause only the first two arguments to be checked.

There is one case where information about unused or undefined variables is more distracting than helpful. This is when *lint* is applied to some but not all files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used. Conversely, many functions and variables defined elsewhere may be used. The **—u** option may be used to suppress the spurious messages which might otherwise appear.

4.3.2 Set/Used Information

The *lint* program attempts to detect cases where a variable is used before it is set. The *lint* program detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use", since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that *lint* can print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialised to zero, no meaningful information can be discovered about their uses. The *lint* program does deal with initialised automatic variables.

The set/used information also permits recognition of those local variables which are set and never used. These form a frequent source of inefficiencies and also be symptomatic of bugs.

4.3.3 Flow of Control

The *lint* program attempts to detect unreachable portions of the programs which it processes. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom and to recognise the special cases **while(1)** and **for (;;)** as infinite loops. The *lint* program also prints messages about loops which cannot be entered at the top. Some valid programs may have such loops which are considered to be bad style at best and bugs at worst.

The *lint* program has no way of detecting functions which are called and never returned. Thus, a call to *exit*(3C) may cause an unreachable code which *lint* does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached but it is not apparent to *lint*, the comment

```
/*NOTREACHED*/
```

can be added at the appropriate place. This comment will inform *lint* that a portion of the program cannot be reached.

The *lint* program will print a message about unreachable **break** statements. Programs generated by *yacc* and especially *lex* may have hundreds of unreachable **break** statements. The **—O** option in the C compiler will often eliminate the

resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do about them, and the resulting messages clutter up the *lint* output. If these messages are not desired, *lint* can be invoked with the **—b** option.

4.3.4 Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function "values" that have never been returned. The *lint* program addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

return(*expr* **);**

and

return;

statements is cause for alarm; the *lint* program will give the message

function *name* contains return(e) and return

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f(a){
    if (a) return (3);
    g ();
}
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a message from *lint*. If *g*, like *exit*(3C), never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, *lint* detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

4.3.5 Type Checking

The *lint* program enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- Across certain binary operators and implied assignments
- At the structure selection operators
- Between the definition and uses of functions
- In the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialisation suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts .

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialisation, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/*NOSTRICT*/
```

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

4.3.6 Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The *lint* program will print a message as a result of detecting this. Consider the assignment

```
p = (char *) 1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signalled the intention. It seems harsh for *lint* to continue to print messages about

this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **—c** flag controls the printing of comments about casts. When **—c** is in effect, casts are treated as though they were assignments subject to messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

4.3.7 Nonportable Character Use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, *lint* will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
if ( (c = getchar()) < 0)...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since *getchar*(3S) is actually returning integer values. In any case, *lint* will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**.

4.3.8 Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedef**s. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **int**s, which are truncated. Since there are a number of legitimate reasons for assigning **long**s to **int**s, the detection of these assignments is disabled by the **—a** option.

4.3.9 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by *lint*. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **—h** option is used to suppress these checks. For example, in the statement

```
*p++;
```

the * does nothing. This provokes the message "null effect" from *lint*. The following program fragment:

```
unsigned x ;
if ( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if ( x > 0 ) ...
```

is equivalent to

```
if ( x != 0 )
```

which may not be the intended action. The *lint* program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if ( 1 != 0 ) ...
```

lint will print the message "constant in conditional context" since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statement

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesise such expressions, and *lint* encourages this by an appropriate message.

Finally, when the **—h** option has not been used, *lint* prints messages about variables which are reduced in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered to be bad style, usually unnecessary, and frequently a bug.

4.3.10 Old Syntax

Several forms of older syntax are now illegal. These fall into two classes — assignment operators and initialisation.

The older forms of assignment operators (eg.=+, =-, ...) could cause ambiguous expressions, such as:

```
a=-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (eg.+=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, *lint* prints messages about these old-fashioned operators.

A similar issue arises with initialisation. The older language allowed

```
int x 1;
```

to initialise x to 1. This also caused syntactic difficulties. For example, the initialisation

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x (y) { ...
```

and the compiler must read past x in order to determine the correct meaning. Again, the problem is even more perplexing when the initialiser involves a macro. The current syntax places an equals sign between the variable and the initialiser:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

4.3.11 Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. The *lint* program tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message "possible alignment problem" results from this situation.

4.3.12 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The *lint* program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause *lint* to print the message "warning: i evaluation order undefined" in order to call attention to this condition.

X/OPEN

PORTABILITY GUIDE

THE X/OPEN COBOL DEFINITION

Contents

Chapter 1

Introduction

The X/OPEN COBOL definition identifies a common set of language facilities that will be supported by COBOL compilers on the X/OPEN systems of the member companies.

The ISO working group and American National Standards COBOL Committee have been working for some years towards a revised standard for COBOL to reflect more accurately the capabilities of modern COBOL compiling systems. The latest international standard for COBOL is defined in "ANS 3.23 - 1985" and "ISO 1989 - 1985", which was approved during 1985. At the time of publication of Issue 2 of the Portability Guide, there were few compilers in compliance with the revised standard and hence the X/OPEN group feels that major changes in the X/OPEN COBOL Definition to reflect the new standard would be premature. It is however likely that a future issue of the X/OPEN COBOL definition will relate to "COBOL 1985", and the new standard has been used as a reference when eliminating obsolete elements from the X/OPEN definition.

The most widely followed standard for COBOL is still that defined in the earlier 1974 Standard "ANS X3.23 - 1974", to which most current COBOL compilers substantially conform.

The 1974 standard is incomplete in the area of facilities for interaction with the on-line user. To overcome this deficiency, most COBOL compilers provide extensions to the *ACCEPT* and *DISPLAY* verbs, but unfortunately they do this in incompatible ways. It is necessary therefore to specify the form of *ACCEPT* and *DISPLAY* to be included in the Common Applications Environment.

In order to have a definition that is achievable on X/OPEN systems within a short timeframe, and one that would immediately have wide acceptance, it has been based on the definition of COBOL embodied in a popular product, Micro Focus LEVEL II COBOL which itself conforms to the "ANS X3.23 - 1974".

The Micro Focus LEVEL II language specification includes enhancements to the ANS standard in addition to the extensions to *ACCEPT* and *DISPLAY*. None of these is currently included in the X/OPEN definition, although they may be supported on specific member systems.

The X/OPEN definition also applies restrictions to the ANS based parts of the LEVEL II definition, particularly elements of the 1974 standard which are becoming obsolete.

Whilst the definition is based on the specification embodied in a particular product, the means of implementation across X/OPEN systems may vary.

In certain cases where the ISO/ANS standard allows an element to be implementer defined, X/OPEN defines a standard implementation across all X/OPEN compliant systems. Where this has been done, the clause is annotated in the syntax definition.

The X/OPEN COBOL definition is given in Chapter 2. It is derived from the Syntax Summary of the LEVEL II COBOL Reference Manual. The semantics of the language are those of the 1974 standard as documented in the Micro Focus LEVEL II COBOL language specification. The LEVEL II facilities that are additional to the ANS/ISO standard and included in the X/OPEN definition are indicated by shading. With the exception of those extensions that are obvious, a full definition of their semantics is contained in Chapter 3.

Chapter 4 summarises the functions in the 1974 standard which are excluded from the X/OPEN definition. These are described in relation to the ANS defined *modules*. This information is included to allow those familiar with the 1974 standard to obtain a quick appreciation of the X/OPEN definition, and it is useful in assessing whether a particular compiler is likely to meet the specification.

Chapter 5 addresses the special characteristics of the System V operating system and the effects these have on the functionality of COBOL compilers, with recommended techniques for ensuring the maximum portability of COBOL programs.

Chapter 2

COBOL Definition

The definition is derived from the Syntax Summary sheets from the Micro Focus LEVEL II COBOL Reference Manual with the following specific notation:

- Shaded areas — indicate areas where the X/OPEN definition deviates from or clarifies the 1974 standard.

  The reason for the shading is indicated in the right margin by:

  E Micro Focus extension included in the X/OPEN definition.

  D Clause which is documentary only.

  I An element which is implementer-defined in the 1974 standard but which is defined by X/OPEN.

The tables use standard COBOL notation to define the language syntax:

- Upper case — is used for COBOL language keywords and also implementation-dependant names which are defined by X/OPEN. Those underlined must be present if the clause is present; those not underlined are "noise words", which may be included to improve readability but are otherwise not processed by the compiler.
- Lower case — strings are generic terms which represent substitutable arguments, for example data names and literal values.
- Square brackets — are used to enclose optional clauses (according to the context); any clause not so enclosed is mandatory.
- Braces — are used to enclose alternatives. One of the alternatives enclosed within the braces must be used. Note that braces and square brackets may be used together to indicate alternative constructs within an optional element. Braces may also be used together with ellipses to delimit a repeatable construct.
- Ellipses... — are used to denote that the preceding element may be repeated a number of times. There must be at least one occurence, unless the element is optional (enclosed in square brackets).

Clauses shown as documentary may be required by certain compilers. Those compilers which do not require the clause to be present will treate it as a comment.

GENERAL FORMAT FOR IDENTIFICATION DIVISION

```
IDENTIFICATION DIVISION.

PROGRAM ID.    program-name.
[AUTHOR.       [comment entry] ...]                    D
[INSTALLATION. [comment entry] ...]                    D
[DATE-WRITTEN. [comment entry] ...]                    D
[DATE-COMPILED.[comment entry] ...]                    D
[SECURITY.     [comment entry] ...]                    D
```

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.
    computer-name [WITH DEBUGGING MODE]

OBJECT-COMPUTER.
    computer-name
    [, MEMORY SIZE integer {WORDS      }]
                           {CHARACTERS }
                           {MODULES    }
    [, PROGRAM COLLATING SEQUENCE IS alphabet-name]
    [, SEGMENT-LIMIT IS segment-number].                    D
```

```
[SPECIAL-NAMES.

[, {SYSIN  } IS mnemonic-name-1]                                          I
   {SYSOUT }

[SWITCH {0}  [IS mnemonic-name]                                           I
        {.}
        {7}
   {ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2]}
   {OFF STATUS IS condition-name-2 [ON STATUS IS condition-name-1]}]

[ , alphabet-name IS
 {STANDARD-1                                                                }
 {NATIVE     literal-1 [{{THROUGH} literal-2                                }]
                        {{THRU   }                                          }
                        {ALSO literal-3 [, ALSO literal 4]...               }
                  [literal-5 [{{THROUGH} literal-6                          }]]...
                             {{THRU   }                                     }
                             {ALSO literal-7 [, ALSO literal 8]...          }
 }...]

[, CURRENCY SIGN IS literal-9]
[, DECIMAL-POINT IS COMMA]
[, CURSOR IS data-name-1]                                                 E
[, CONSOLE IS CRT]                                                        E
[, CRT STATUS IS data-name-2]                                             E
].
```

```
[ INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    [ {file-control-entry}...
      I-O-CONTROL.
      [ ; SAME [RECORD] AREA FOR
                        file-name-3 [, file-name-4] ... ] ... . ]
```

GENERAL FORMAT FOR FILE-CONTROL ENTRY

SEQUENTIAL SELECT:

```
SELECT [OPTIONAL] file-name

ASSIGN TO { path-name-literal-1    }                                  E
          { implementor-name-1     }
        [ , { path-name-literal-2  } ] ...                           D
            { implementor-name-2   }

    [ ; RESERVE integer-1 { AREA  } ]                                D
                          { AREAS }

    [; ORGANIZATION IS SEQUENTIAL]

    [; ACCESS MODE IS SEQUENTIAL]

    [; FILE STATUS IS data-name].
```

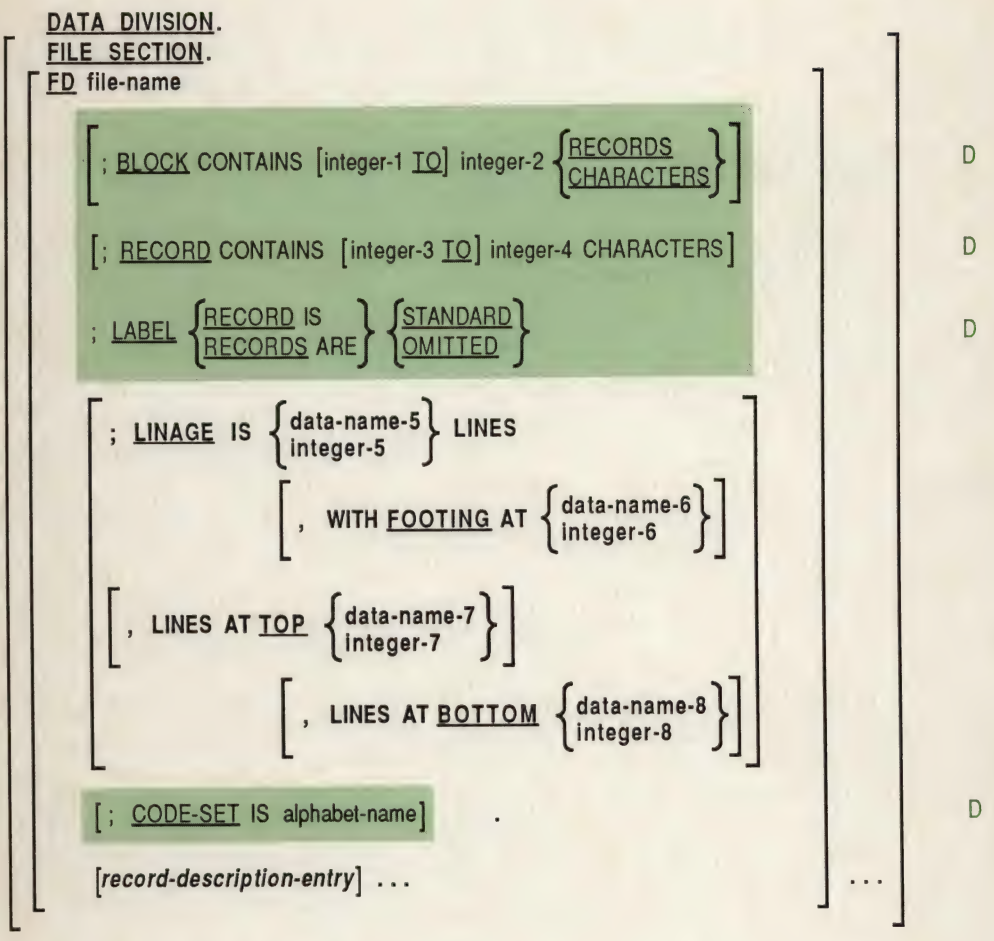
RELATIVE SELECT:

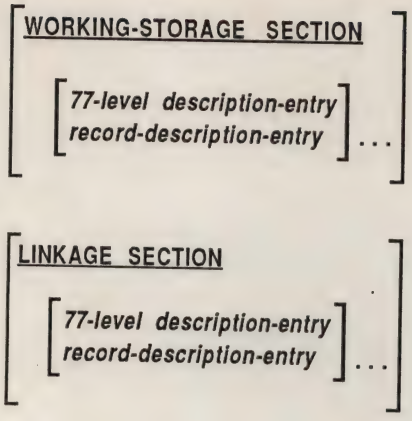
```
SELECT file-name

ASSIGN TO { path-name-literal-1     }                                    E
          { implementor-name-1      }
        [ , { path-name-literal-2   } ] ...                              D
            { implementor-name-2    }

[ ; RESERVE integer-1 { AREA  } ]                                        D
                      { AREAS }

  ; ORGANIZATION IS RELATIVE

[ ; ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name] }
                   { RANDOM     , RELATIVE KEY IS data-name   }
                   { DYNAMIC                                  } ]

[; FILE STATUS IS data-name].
```

INDEXED SELECT:

```
SELECT file-name

ASSIGN TO { path-name-literal-1     }                                    E
          { implementor-name-1      }
        [ , { path-name-literal-2   } ] ...                              D
            { implementor-name-2    }

[ ; RESERVE integer-1 { AREA  } ]                                        D
                      { AREAS }

  ; ORGANIZATION IS INDEXED

[ ; ACCESS MODE IS { SEQUENTIAL } ]
                   { RANDOM     }
                   { DYNAMIC    }

  ; RECORD KEY IS data-name-1

[ ; ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES] ] ...

[ ; FILE STATUS IS data-name-3] .
```

GENERAL FORMAT FOR THE DATA DIVISION




WORKING-STORAGE SECTION
77-level description-entry
record-description-entry
...
LINKAGE SECTION
77-level description-entry
record-description-entry
...

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

FORMAT 1:

```
level-number { data-name-1 }
             { FILLER      }

    [; REDEFINES data-name-2]

    [; { PICTURE } IS picture-string]
       { PIC     }

    [; [USAGE IS] { COMPUTATIONAL   }]                                  E
                  { COMP            }
                  { COMPUTATIONAL-3 }                                   E
                  { COMP-3          }
                  { DISPLAY         }
                  { INDEX           }

    [; [SIGN IS] { LEADING  } [SEPARATE CHARACTER]]
                 { TRAILING }

    [; OCCURS
         { integer-1 TO integer-2 TIMES DEPENDING ON data-name-3 }
         { integer-2 TIMES                                       }

         [ { ASCENDING  } KEY IS data-name-4 [, data-name-5]... ]...
           { DESCENDING }

           [INDEXED BY index-name-1 [, index-name-2]... ]      ]

    [; { SYNCHRONIZED } [ { LEFT  } ]]                                  D
       { SYNC         }   { RIGHT }

    [; { JUSTIFIED } RIGHT]
       { JUST      }

    [; BLANK WHEN ZERO]

    [; VALUE IS literal].
```

FORMAT 2:

```
66 data-name-1; RENAMES data-name-2 [ {THROUGH} data-name-3 ].
                                    {THRU   }
```

FORMAT 3:

```
88 condition-name ; {VALUE IS  } literal-1 [ {THROUGH} literal-2 ]
                   {VALUES ARE}             {THRU   }

                   [ , literal-3 [ {THROUGH} literal-4 ] ]... .
                                 {THRU   }
```

GENERAL FORMAT FOR PROCEDURE DIVISION

FORMAT 1:

```
PROCEDURE DIVISION
    [USING data-name-1 [, data-name-2] ... ].

[ DECLARATIVES .                                                          ]
[   { section-name SECTION [segment-number]. declarative-sentence         }  D
[     [ paragraph-name. [sentence] ... ] ... } ...                        ]
[ END DECLARATIVES .                                                      ]

  { section-name SECTION [segment-number].                                   D
    [ paragraph-name. [sentence] ... ] ... } ...
```

FORMAT 2:

```
PROCEDURE DIVISION
    [USING data-name-1 [, data-name-2] ... ].

    { paragraph-name. [sentence] ... } ...
```

GENERAL FORMAT FOR ACCEPT STATEMENT

```
ACCEPT identifier [FROM mnemonic-name]

ACCEPT data-name-1 { [ AT { data-name-2 } ] FROM CRT }                    E
                   {       { literal-1  }                }
                   { AT { data-name-2 }                  }
                   {    { literal-1  }                   }

ACCEPT identifier FROM { DATE }
                       { DAY  }
                       { TIME }
```

GENERAL FORMAT FOR ADD STATEMENT

```
ADD { identifier-1 } [ , identifier-2 ] ...
    { literal-1    }   [ , literal-2   ]

    TO identifier-m [ROUNDED]

    [ , identifier-n [ROUNDED] ] ...

    [; ON SIZE ERROR imperative statement]
```

```
ADD { identifier-1 } , { identifier-2 } [ , identifier-3 ] ...
    { literal-1    }   { literal-2    }   [ , literal-3   ]

    GIVING identifier-m [ROUNDED] [ , identifier-n [ROUNDED] ] ...

    [; ON SIZE ERROR imperative statement]
```

```
ADD { CORRESPONDING } identifier-1 TO identifier-2 [ROUNDED]
    { CORR          }

    [; ON SIZE ERROR imperative statement]
```

GENERAL FORMAT FOR CALL STATEMENT

```
CALL literal-1 [USING data-name-1 [, data-name-2]... ]
```

GENERAL FORMAT FOR CLOSE STATEMENT

```
CLOSE file-name-1 [ { REEL } [WITH NO REWIND]          ]   D
                  { UNIT } [FOR REMOVAL      ]
                  [ WITH { NO REWIND } ]
                         { LOCK      }
                  [
                  , file-name-2 [ { REEL } [WITH NO REWIND]          ]   D
                                { UNIT } [FOR REMOVAL      ]
                                [ WITH { NO REWIND } ]
                                       { LOCK      }
                  ]...
```

GENERAL FORMAT FOR COMPUTE STATEMENT

```
COMPUTE identifier-1 [ROUNDED] [, identifier-2 [ROUNDED] ] ...
        = arithmetic-expression
        [; ON SIZE ERROR imperative statement]
```

GENERAL FORMAT FOR COPY STATEMENT

```
COPY { text-name { OF } library-name }
     {           { IN }              }
     { path-name-literal             }

[ REPLACING { , { = =pseudo-text-1= = }  BY { = =pseudo-text-2= = } } ... ]
            {   { identifier-1         }     { identifier-2         } }
            {   { literal-1            }     { literal-2            } }
            {   { word-1               }     { word-2               } }
```

E

GENERAL FORMAT FOR DELETE STATEMENT

```
DELETE file-name RECORD [; INVALID KEY Imperative-statement]
```

GENERAL FORMAT FOR DISPLAY STATEMENT

```
DISPLAY { identifier-1 } [ , identifier-2 ] ... [UPON mnemonic-name]
        { literal-1    }   { , literal-2    }

DISPLAY { data-name-1 } { [ AT { data-name-2 } ] UPON { CRT      } }
        { literal-3   } {      { literal-4   }        { CRT-UNDER } }
                        {   AT { data-name-2 }                     }
                        {      { literal-4   }                     }
```

E

GENERAL FORMAT FOR DIVIDE STATEMENT

```
DIVIDE { identifier-1 }
       { literal-1    }

    INTO identifier-2 [ROUNDED]

    [, identifier-3 [ROUNDED] ] ...

    [; ON SIZE ERROR imperative-statement]
```

```
DIVIDE { identifier-1 } { INTO } { identifier-2 }
       { literal-1    } { BY   } { literal-2    }

    GIVING identifier-3 [ROUNDED]

    [, identifier-4 [ROUNDED] ] ...

    [; ON SIZE ERROR imperative-statement]
```

```
DIVIDE { identifier-1 } { INTO } { identifier-2 }
       { literal-1    } { BY   } { literal-2    }

    GIVING identifier-3 [ROUNDED]

    REMAINDER identifier-4

    [; ON SIZE ERROR imperative-statement]
```

GENERAL FORMAT FOR EXIT STATEMENT

```
EXIT [PROGRAM].
```

GENERAL FORMAT FOR GO TO STATEMENT

```
GO TO procedure-name

GO TO procedure-name-1 {, procedure-name-2}...
     DEPENDING ON identifier
```

GENERAL FORMAT FOR IF STATEMENT

```
IF condition; { statement-1      }
              { NEXT SENTENCE    }

     [ ; ELSE { statement-2      } ]
              { NEXT SENTENCE    }
```

GENERAL FORMAT FOR INSPECT STATEMENT

INSPECT identifier-1 **TALLYING** *tally-clause*

INSPECT identifier-1 **REPLACING** *replacing-clause*

INSPECT identifier-1 **TALLYING** *tally-clause* **REPLACING** *replacing-clause*

TALLY CLAUSE

```
{ identifier-2 FOR { { { ALL     } { identifier-3 } }                                          }
                   { { { LEADING } { literal-2    } } CHARACTERS                              }
                   {                                                                          }
                   {   [ { BEFORE } INITIAL { identifier-4 } ] } ... }
                   {   [ { AFTER  }         { literal-3    } ]     }
                                                                                          ...
```

REPLACING CLAUSE

```
{ CHARACTERS BY { identifier-6 } [ { BEFORE } INITIAL { identifier-7 } ]                                                  }
               { literal-4    }   { AFTER  }         { literal-5    }                                                      }
  { , { ALL     }   { , { identifier-5 } BY { identifier-6 }                                                            }
      { LEADING }       { literal-3    }    { literal-4    }                                                            }
      { FIRST   }                                                                                                       }
                  [ { BEFORE } INITIAL { identifier-7 } ] } ... }                                                       }
                  { AFTER  }         { literal-5    }   }     }                                                       }
                                                                                                                   ...
```

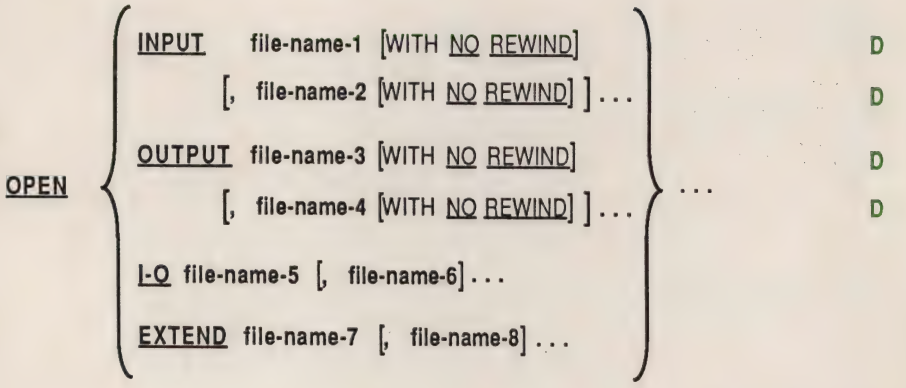
GENERAL FORMAT FOR MOVE STATEMENT

```
MOVE { identifier-1 } TO identifier-2 [, identifier-3] ...
     { literal-1    }

MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR          }
```

GENERAL FORMAT FOR MULTIPLY STATEMENT

```
MULTIPLY { identifier-1 }
         { literal-1    }

    BY identifier-2 [ROUNDED]

    [, identifier-3 [ROUNDED]] ...

    [; ON SIZE ERROR imperative-statement]

MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }    { literal-2    }

    GIVING identifier-3 [ROUNDED] [, identifier-4 [ROUNDED] ] ...

    [; ON SIZE ERROR imperative-statement]
```

GENERAL FORMAT FOR OPEN STATEMENT



GENERAL FORMAT FOR PERFORM STATEMENT

```
PERFORM procedure-name-1
    [ { THROUGH } procedure-name-2 ]
      { THRU    }

PERFORM procedure-name-1
    [ { THROUGH } procedure-name-2 ] { identifier-1 } TIMES
      { THRU    }                    { integer-1    }

PERFORM procedure-name-1
    [ { THROUGH } procedure-name-2 ] UNTIL condition-1
      { THRU    }

PERFORM procedure-name-1
    [ { THROUGH } procedure-name-2 ]
      { THRU    }

    VARYING { identifier-2 } FROM { identifier-3 }
            { index-name-1 }      { index-name-2 }
                                  { literal-1    }

        BY { identifier-4 } UNTIL condition-1
           { literal-2    }

  [ AFTER { identifier-5 } FROM { identifier-6 }
          { index-name-3 }      { index-name-4 }
                                { literal-3    }

      BY { identifier-7 } UNTIL condition-2
         { literal-4    }

    [ AFTER { identifier-8 } FROM { identifier-9 }
            { index-name-5 }      { index-name-6 }
                                  { literal-5    }

      BY { identifier-10 } UNTIL condition-3
         { literal-6     }                    ]  ]
```

GENERAL FORMAT FOR READ STATEMENT

SEQUENTIAL ORGANISATION:

READ file-name **RECORD** [**INTO** identifier]

[; **AT END** *imperative-statement*]

RELATIVE ORGANISATION:

READ file-name [**NEXT**] **RECORD** [**INTO** identifier]

[; **AT END** *imperative-statement*]

READ file-name **RECORD** [**INTO** identifier]

[; **INVALID KEY** *imperative-statement*]

INDEXED ORGANISATION:

READ file-name [**NEXT**] **RECORD** [**INTO** identifier]

[; **AT END** *imperative-statement*]

READ file-name **RECORD** [**INTO** identifier]

[; **KEY IS** data-name]

[; **INVALID KEY** *imperative-statement*]

GENERAL FORMAT FOR REWRITE STATEMENT

```
REWRITE record-name [FROM identifier]
    [; INVALID KEY imperative-statement]
```

GENERAL FORMAT FOR SEARCH STATEMENT

```
SEARCH identifier-1 [ VARYING { identifier-2 } ]
                              { index-name-1 }
    [; AT END imperative-statement-1 ]
    ; WHEN condition-1 { imperative-statement-2 }
                       { NEXT SENTENCE          }
  [ ; WHEN condition-2 { imperative-statement-3 } ] ...
                       { NEXT SENTENCE          }

SEARCH ALL identifier-1 [; AT END imperative-statement-1 ]
    ; WHEN { data-name-1      { IS EQUAL TO } { identifier-3          } }
           {                  { IS =        } { literal-1             }
           {                                  { arithmetic-expression-1 }
           { condition-name-1                                           }

  [ AND { data-name-1      { IS EQUAL TO } { identifier-4          } } ]
        {                  { IS =        } { literal-2             }
        {                                  { arithmetic-expression-2 }
        { condition-name-2                                           }

    { imperative-statement-2 }
    { NEXT SENTENCE          }
```

GENERAL FORMAT FOR SET STATEMENT

```
SET { identifier-1  [identifier-2] }  ...  TO { identifier-3 }
    { index-name-1  [index-name-2] }          { index-name-3 }
                                              { integer-1    }

SET index-name-4 [, index-name-5] ... { UP BY   } { identifier-4 }
                                      { DOWN BY } { integer-2    }
```

GENERAL FORMAT FOR START STATEMENT

```
START file-name [ KEY { IS EQUAL                  } data-name ]
                      { IS =                      }
                      { IS GREATER THAN           }
                      { IS >                      }
                      { IS NOT LESS THAN          }
                      { IS NOT <                  }

    [; INVALID KEY imperative-statement]
```

GENERAL FORMAT FOR STOP STATEMENT

```
STOP { RUN     }
     { literal }
```

GENERAL FORMAT FOR STRING STATEMENT

```
STRING { identifier-1 } [ , { identifier-2 } ] ... DELIMITED BY { identifier-3 }
       { literal-1    }     { literal-2    }                    { literal-3    }
                                                               { SIZE         }

    [ , { identifier-4 } [ , { identifier-5 } ] ... DELIMITED BY { identifier-6 } ] ...
        { literal-4    }     { literal-5    }                    { literal-6    }
                                                               { SIZE         }

    INTO identifier-7 [WITH POINTER identifier-8]

    [, ON OVERFLOW imperative-statement]
```

GENERAL FORMAT FOR SUBTRACT STATEMENT

```
SUBTRACT { identifier-1 } [ , { identifier-2 } ] ...
         { literal-1    }     { literal-2    }

    FROM identifier-m [ROUNDED] [ , identifier-n [ROUNDED] ] ...

    [ ; ON SIZE ERROR imperative-statement ]

SUBTRACT { identifier-1 } [ , { identifier-2 } ] ...
         { literal-1    }     { literal-2    }

    FROM { identifier-m }
         { literal-m    }

    GIVING identifier-n [ROUNDED] [ , identifier-o [ROUNDED] ] ...

    [ ; ON SIZE ERROR imperative-statement ]

SUBTRACT { CORRESPONDING } identifier-1
         { CORR          }

    FROM identifier-2 [ROUNDED]

    [ ; ON SIZE ERROR imperative-statement ]
```

GENERAL FORMAT FOR UNSTRING STATEMENT

```
UNSTRING identifier-1

[ DELIMITED BY [ALL] { identifier-2 } [ , OR [ALL] { identifier-3 } ] ... ]
                     { literal-1   }               { literal-2   }

  INTO identifier-4 [, DELIMITER IN identifier-5] [, COUNT IN identifier-6]

    [, identifier-7 [, DELIMITER IN identifier-8] [, COUNT IN identifier-9]] ...

  [WITH POINTER identifier-10] [TALLYING IN identifier-11]

  [; ON OVERFLOW imperative-statement]
```

GENERAL FORMAT FOR USE STATEMENT

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE
                   { ERROR     }

    ON { file-name-1 [, file-name-2] ... }
       { INPUT                          }
       { OUTPUT                         }
       { I-O                            }
       { EXTEND                         }
```

GENERAL FORMAT FOR WRITE STATEMENT

SEQUENTIAL ORGANISATION:

```
WRITE record-name [FROM identifier-1]

[ {BEFORE} ADVANCING { {integer   } [LINE ] }  ]
[ {AFTER }           { {identifier-2} [LINES] }  ]
                     {  PAGE                  }

[; AT {END-OF-PAGE} imperative-statement ]
      {EOP        }
```

RELATIVE AND INDEXED ORGANISATION:

```
WRITE record-name [FROM identifier]

[; INVALID KEY imperative-statement]
```

GENERAL FORMAT FOR CONDITIONS

Relation condition:

```
{identifier-1                }  {IS [NOT] GREATER THAN          }  {identifier-2                }
{literal-1                   }  {IS [NOT] LESS THAN             }  {literal-2                   }
{arithmetic-expression-1     }  {IS [NOT] EQUAL TO              }  {arithmetic-expression-2     }
{index-name-1                }  {IS [NOT] >                     }  {index-name-2                }
                                {IS [NOT] <                     }
                                {IS [NOT] =                     }
```

Class condition:

```
identifier IS [NOT] {NUMERIC   }
                    {ALPHABETIC}
```

Sign condition:

```
arithmetic-expression IS [NOT] {POSITIVE}
                               {NEGATIVE}
                               {ZERO    }
```

Condition-name condition:

condition-name

Switch-status condition:

condition-name

Negated simple condition:

NOT simple-condition

Combined condition:

condition { { **AND** / **OR** } **condition** } ...

Abbreviated combined relation condition:

relation-condition { { **AND** / **OR** } **[NOT] [relational-operator] object** } ...

MISCELLANEOUS FORMATS

QUALIFICATION:

```
{data-name-1     } [ {OF} data-name-2 ] ...
{condition-name  }   {IN}

paragraph-name [ {OF} section-name ]
               {IN}

text-name [ {OF} library-name ]
          {IN}
```

SUBSCRIPTING:

```
{data-name     } (subscript-1 [, subscript-2 [, subscript-3 ] ] )
{condition-name}
```

INDEXING:

```
{data-name      } ( {index-name-1 [{+} literal-2]}
{condition-name }   {literal-1    [{-}         ]}

    [ , {index-name-2 [{+} literal-4]}
        {literal-3    [{-}         ]}

      [ , {index-name-3 [{+} literal-6]}
          {literal-5    [{-}         ]} ] ]  )
```

IDENTIFIER: FORMAT 1

```
data-name-1 [ {OF} data-name-2 ] ...
              {IN}

    [ ( subscript-1 [, subscript-2 [, subscript-3]] ) ]
```

IDENTIFIER: FORMAT 2

```
data-name-1 [ { OF } data-name-2 ] ...
            [ { IN }             ]

[ ( { index-name-1 [ { + } literal-2 ] } )                                    ]
    { literal-1    [ { - }           ] }                                      ]
[ , { index-name-2 [ { + } literal-4 ] }                                      ]
    { literal-3    [ { - }           ] }                                      ]
[ , { index-name-3 [ { + } literal-6 ] } ] ] ]                                ]
    { literal-5    [ { - }           ] }                                      ]
```

Chapter 3

Definition of Extensions

The specification of ANS COBOL (1974) is documented in "ANS 3.23 - 1974", so there is little value in repeating such detail in this Guide. However, those elements of Micro Focus LEVEL II COBOL which have been defined as part of the X/OPEN definition, namely the *ACCEPT* and *DISPLAY* verbs are less widely published.

This chapter contains a full definition of the *ACCEPT* and *DISPLAY* verbs, together with the necessary extensions to *SPECIAL-NAMES*. The syntax definition for each is given in full, including both those elements which are part of the ANS standard and the X/OPEN extensions. Full semantics are given only for the X/OPEN extensions. As in the previous chapter, X/OPEN extensions are shaded.

SPECIAL-NAMES(COBOL)

NAME

SPECIAL-NAMES — provides a means of relating system specific names to user-specific mnemonic names and of relating alphabet-names to character sets and/or collating sequences

SYNOPSIS

```
[SPECIAL-NAMES.

[, {SYSIN  } IS mnemonic-name-1]                                          I
   {SYSOUT }

[SWITCH {0} [IS mnemonic-name]                                            I
        {.}
        {.}
        {7}
    {ON STATUS IS condition-name-1 [OFF STATUS IS condition-name-2]}
    {OFF STATUS IS condition-name-2 [ON STATUS IS condition-name-1]}]

[ , alphabet-name IS
 {STANDARD-1                                                                }
 {NATIVE      literal-1 [{{THROUGH} literal-2                               }
 {                       {THRU   }                                          }
 {                       ALSO literal-3 [, ALSO literal 4]...]              }
 {        [literal-5 [{{THROUGH} literal-6                                  }
 {                   {THRU   }                                             }
 {                   ALSO literal-7 [, ALSO literal 8]}...]]...            }...]

[, CURRENCY SIGN IS literal-9]
[, DECIMAL-POINT IS COMMA]
[, CURSOR IS data-name-1]                                                 E
[, CONSOLE IS CRT]                                                        E
[, CRT STATUS IS data-name-2]                                             E
].
```

DESCRIPTION

1 *Mnemonic-names* can be any COBOL user-defined word and at least one constituent character must be alphabetic.

2 *data-name-2* must be defined in Data Division as an alphanumeric data item, three characters in length.

3 *data-name-2* may be a group item.

4 The clause CONSOLE IS changes the defaults in the ACCEPT and DISPLAY statements.

5 The clause CURSOR IS *data-name-1* specifies the data-name to contain the cursor address as used by the ACCEPT statement. If CURSOR IS is not specified, the default cursor position on executing an ACCEPT statement is the "home" position at top left of the CRT screen. The CURSOR IS clause enables a program to retain the position at the end of the exectution of the last ACCEPT statement or to specify the intial position at the start of any ACCEPT statement.

data-name-1 contains the name either of a PIC 9999 field in which the most significant 99 represents a line count in the range one to the maximum number of lines on the user screen, and the least significant 99 represents a character position in the range one to the maximum number of positions allowed by the user screen; or of a PIC 9(6) field in which the most significant 999 represents the line count and the least significant 999 the character position (thus allowing use of screens wider than 99 characters).

6 The SYSIN clause renames the system logical input unit (that is, the CRT keyboard).

7 The SYSOUT clause renames the system logical output unit (that is, the CRT screen).

8 If the CRT STATUS IS clause is specified, a value will be moved into the data item specified by *data-name-2* after each ACCEPT FROM CRT statement is executed. Full details of the value to be contained in the data-item are given in the definition of ACCEPT.

ACCEPT(COBOL)

NAME

ACCEPT — causes data keyed at the CRT console to be made available to the program in a specified data item.

SYNOPSIS

Format 1

```
ACCEPT identifier [FROM mnemonic-name]
```

Format 2

```
ACCEPT data-name-1 { [ AT { data-name-2 } ] FROM CRT }
                   { [    { literal-1  } ]           }
                   {   AT { data-name-2 }            }
                   {      { literal-1  }             }
```

E

Format 3

```
ACCEPT identifier FROM { DATE }
                       { DAY  }
                       { TIME }
```

DESCRIPTION

Syntax Rule

The *mnemonic-name* in Format 1 must also be specified in the SPECIAL-NAMES paragraph of the Environment Division and must be associated with the console.

General Rules

Format 1 is the standard ANS/ISO ACCEPT statement. If the FROM phase is omitted, the default assumes SYSIN. The default can, however, be changed by specifying CONSOLE IS CRT in the SPECIAL-NAMES paragraph so that FROM CRT becomes the default.

Format 2 is the extended ACCEPT format.

Format 2

1 The ACCEPT statement causes the transfer of data from the CRT to *data-name-1*. The contents of *data-name-1* are replaced by this data.

2 *data-name-1* is taken as a definition of the screen area in which elementary data items correspond to areas on the screen into which the operator can key. FILLER fields correspond to areas on the screen which are inaccessible to the operator. *data-name-1* must not be subscripted.

3 Elementary data items within *data-name-1* may be alphanumeric, numeric, usage display, or edited. Non-integer numeric items are treated as two separate integer numeric fields, and edited fields are treated as alphanumeric fields except as described below in rule 11.

4 AT *data-name-2* or *literal-1* defines the position on the screen of the leftmost character of the data. Either form must refer to a PIC 9999 or PIC 9(6) field. The most significant half is taken as the line count in the range one to the maximum lines on the user screen. The least significant half is taken as a character position in the range one to the maximum positions allowed by the screen width of the user CRT. Note that the size of this data item is independent of the size of the CURSOR IS data item.

5 *data-name-1* may refer to a record, group or elementary-item, but it may not be subscripted. REDEFINES may be used within *data-name-1*, in which case the first description of the data is used, and subsequent descriptions are ignored. OCCURS and nested OCCURS may also be used with the effect that the repeated data item is expanded into the full number of times it occurs and one definition is thus automatically repeated for many fields.

6 Immediately on execution of the ACCEPT statement, a cursor is displayed in the CRT location corresponding to the leftmost non-FILLER character position in *data-name-1*. The precise location is dependent on the contents and type of this field. Alternatively, when CURSOR is specified in the SPECIAL-NAMES paragraph, the cursor displays at the position held in the CURSOR *data-name*. The CURSOR position is stored in CURSOR *data-name* in the same format as the screen position is held in *data-name-2*. If the CURSOR *data-name* has the value SPACE or ZERO, the effect is as if CURSOR was not specified and the CURSOR *data-name* will not thereafter be updated. If a valid screen position is specified that is not within a non-FILLER item, the cursor is positioned to the next non-FILLER character position. If that position is beyond the final non-FILLER item, the cursor is positioned at the beginning of the first non-FILLER item of the record being accepted. If the value held in the CURSOR data item defines a position in a numeric edited field, the precise position of the cursor depends on the contents of the field. The CURSOR *data-name* holds the last cursor position at the end of execution of an ACCEPT statement.

7 If neither AT ..., FROM CRT nor CONSOLE IS CRT (in SPECIAL-NAMES) is specified, the default assumes SYSIN (see Format 1).

8 As the operator keys characters, the cursor moves to the right one character position at a time in locations corresponding to data fields. The operator always keys into the current cursor position. At the end of a line the cursor always moves down one line and to the leftmost non-FILLER character position.

9 If the data item is integer numeric, only numeric characters (0-9) will be accepted into that item. Keying the decimal point character (. or , as specified in the DECIMAL-POINT phrase) when accepting a numeric item causes the item to be right justified and zero-filled from the left.

10 When the cursor location reaches a position corresponding to a FILLER item in a data-name, it immediately skips to the next non-FILLER character position, or if there is no such position remaining in the portion of the CRT specified by the data-name it

remains in its current position. Any characters then entered will replace the last character.

11 The operator can terminate input by pressing the appropriate key (usually the carriage return <CR>) at which time control is passed to the next statement after ACCEPT. Before control is passed to the next statement the following takes place :

a. The numeric value of each numeric-edited data-field is formed internally from only the keyed characters 0 to 9, +, —, . or , and then moved back to the numerically-edited field with the ANS PICTURE editing applied. The field may thus be different from that shown on the CRT just before the carriage return key was pressed.

b. When CURSOR IS is specified in the SPECIAL-NAMES paragraph, the cursor position when the Carriage Return key is pressed is returned in the data name specified by the CURSOR IS clause, except when its value at the start of the ACCEPT function causes it to be treated as unspecified.

12 Before keying Carriage Return, the operator can reposition the cursor to overwrite data already keyed or to skip character positions by use of the cursor positioning keys.

13 If the CRT STATUS IS *data-name* clause has been used in the SPECIAL-NAMES paragraph of the Environment Division, the data item identified by the clause will be updated after every ACCEPT statement. The contents of this data item and their meanings are:

1st Character

0 Normal termination of the ACCEPT, either by the RETURN key or equivalent or, if the functionality is available, by tabbing out of the last field. Character 2 is undefined.

1 A user-defined function key was used to terminate the ACCEPT. Character 2 contains a COMP number (between 0 and 255) indicating which key was used.

9 An error has occurred. A COMP number in Character 2 determines the error as follows:

0 a null ACCEPT has occurred, for example a "clear screen" terminated the ACCEPT;

8 a character that has been disabled has been keyed;

9 an invalid keystroke (more than one byte) has occurred.

Character 3 is not currently defined

NAME

DISPLAY — causes data to be transferred from specified data items to the CRT screen.

SYNOPSIS

Format 1

```
DISPLAY { identifier-1 } [ , identifier-2 ] ... [UPON mnemonic-name]
        { literal-1    }   { , literal-2   }
```

Format 2

```
DISPLAY { data-name-1 } { [ AT { data-name-2 } ] UPON { CRT       } }
        { literal-3   } {      { literal-4   }        { CRT-UNDER } }
                        {   AT { data-name-2 }                      }
                        {      { literal-4   }                      }
```

E

DESCRIPTION

Syntax Rules

1 The *mnemonic-name* in Format-1 must be associated with the console in the SPECIAL-NAMES paragraph in the Environment Division.

2 Each *literal* may be any figurative constant, except ALL.

3 If the *literal* is numeric, it must be unsigned integer.

4 *literal-3* must be alphanumeric; *literal-4* must be numeric.

0 *data-name-1* may refer to a record, group or elementary item, but it must not be subscripted.

General Rules

1 Format 1 is the standard ANS/ISO DISPLAY statement. If the UPON phrase is omitted, the default assumes SYSOUT. The default can, however, be changed by specifying CONSOLE IS CRT in the SPECIAL-NAMES paragraph so that UPON CRT becomes the default.

2 Format 2 is the extended DISPLAY format.

Format 2

1 The DISPLAY statement is used to output data to the CRT in the screen positions specified.

2 *data-name-1* is taken as a definition of the screen area into which data items that correspond to areas on the screen are moved. FILLER fields correspond to areas on the screen into which data is not moved.

3 *data-name-1* may be an elementary item or may be a group item containing group and/or elementary items. Such elementary items must be defined as USAGE DISPLAY.

4 AT *data-name-2* or *literal-4* defines the position on the screen of the leftmost character of the data. Either form must refer to a PIC 9999 or a PIC 9(6) field. The most significant half is taken as a line count in the range one to the maximum number of lines on the screen. The least significant half is taken as a character position in the range one to the maximum number of characters per line on the user screen.

5 *data-name-1* may refer to a record, group or elementary item, but it may not be subscripted. REDEFINES may be used, in which case the first description of the data is used and subsequent descriptions are ignored. OCCURS and nested OCCURS may be used with the effect that the repeated *data-item* is expanded into the full number of times it occurs and one definition is thus automatically repeated for many fields.

6 DISPLAY SPACE has the effect of clearing the screen at run time (i.e. filling the whole screen with spaces). DISPLAY " " (one space character), however, displays only one space character.

7 The CRT-UNDER phrase causes the elementary items moved to the CRT to be displayed with the underline feature present. This feature is dependant on the CRT hardware functions and is not available on all makes of CRT.

8 If neither AT, UPON CRT, UPON CRT-UNDER nor CONSOLE IS CRT (in SPECIAL-NAMES) is specified, the default assumes SYSOUT (Format 1).

Chapter 4

Summary of Exclusions

This Chapter summarises the exclusions from the 1974 COBOL Standard in a convenient form for those familiar with that document. It describes the exclusions in relation to the *modules* defined in the COBOL standard. The information will be useful for assessing whether a particular compiler meets the X/OPEN definition.

- Complete modules that are in the 1974 standard but are excluded from the X/OPEN definition:

  Report Writer
  Communication
  Sort-Merge

- Modules included by X/OPEN at a lower level (as defined in the 1974 standard). The following list identifies the items excluded:

  Debug
  DEBUG-ITEM
  USE FOR DEBUGGING statement

  Inter-Program Communication
  CALL statement *identifier-1* option
  ON OVERFLOW phrase of *CALL* statement
  CANCEL statement in entirety

- Individual elements included in the 1974 standard but excluded from the X/OPEN definition:

  ALTER statement (Nucleus module)

  DATA clause (Sequential, relative and indexed I/O modules)

  ENTER statement (Nucleus module)

  GO TO statement without *procedure-name* (Nucleus module)

  MULTIPLE FILE TAPE clause (Sequential I/O module)

  OPEN statement phrase (Sequential I/O module):
  REVERSED

  RERUN clause (Sequential, relative and indexed I/O modules)

  VALUE clause (Sequential, relative and indexed I/O modules)

- Individual elements included in the 1974 Standard, but defined as "documentary only" in the X/OPEN definition:

  ellipses in *ASSIGN* clause (Sequential, relative and indexed I/O modules)

BLOCK CONTAINS clause (Sequential, relative and indexed I/O modules)

CLOSE statement phrase (Sequential I/O module)
- *FOR REMOVAL*
- *REEL*
- *UNIT*
- *WITH NO REWIND*

CODE-SET clause (Sequential I/O module)

DATE-COMPILED clause (Nucleus module)

LABEL clause (Sequential, relative and indexed I/O modules)

OPEN statement phrase (Sequential I/O module)
- *WITH NO REWIND*

RECORD CONTAINS clause (Sequential, relative and indexed I/O modules)

RESERVE clause (Sequential, relative and indexed I/O modules)

SEGMENT-LIMIT clause (Segmentation module)

segment-number in *SECTION* header (Segmentation module)

SYNCHRONISED clause (Nucleus module)

Chapter 5

COBOL in a System V Environment

5.1 INTRODUCTION

This chapter addresses the special characteristics of the System V operating system and the effects these have on the functionality of COBOL compilers. It includes recommended techniques for ensuring the maximum portability of COBOL programs.

5.2 FILE ASSIGNMENT AND REASSIGNMENT

In the 1974 standard, the interpretation of the argument of the SELECT ASSIGN clause is implementor-defined.

The basic structure of the clause as defined in the X/OPEN definition is :

SELECT file-name ASSIGN TO { path-name | implementor-name }

In some implementations, the only valid argument is a System V file identifier defined relative to the current directory or the / root directory of the file system. Other implementations allow the argument to be a logical name which is assigned to a physical file at run-time.

Application developers are recommended to adopt the use of names relative to the current directory to facilitate implementation in a variety of systems and not to rely on the availability of any facilities to re-assign at run-time.

5.3 SPECIAL FILES

The X/OPEN COBOL definition includes provision for special files SYSIN/SYSOUT and CRT.

Application developers are recommended to associate standard input/standard output with the System V standard input and standard output devices (stdin and stdout) and CRT with the users terminal.

In the case where the System V standard input and standard output devices are not redirected, a common file descriptor should be used for SYSIN/SYSOUT and CRT. This will allow parameter records to be input from a redirected standard input file without interference with data entered from the CRT. Likewise, data may be output to a redirected output file without interference from data output to the CRT.

The handling of command line arguments differs between implementations of COBOL run-time systems. Application developers are recommended to assume that command line arguments are not available to the COBOL program.

5.4 FILTERS

In certain circumstances it is useful to read the input to a COBOL program and write the output from a COBOL program from/to a pipe. This is only available for SEQUENTIAL file organisations; OPEN FOR I-O would cause a run-time error.

5.5 MEMORY ORGANISATION

The manner in which memory is allocated for CALLed modules may differ between interpreted and fully compiled implementations. The management of memory required for a series of CALLed modules should not be defined by the application developer but dynamically at run-time. This method of memory allocation will offer maximum portability but could cause memory fragmentation. Additionally, developers should not make any assumptions regarding the contents of variables on second and subsequent calls to a module.

5.6 COPY LIBRARY LOCATION

The constraints on the location of the COPY library members differ from compiler to compiler. Application developers should assume that the only way of defining such a location is by the definition of a System V file identifier relative to the current directory or the system / (root) directory. The ability to define the name of a directory in the "OF library-name" clause should not be assumed.

5.7 ERROR REPORTING

The default for the output of both compile-time and run-time errors should be to the (possibly redirected) standard error device.

ISBN: 0 444 70177 X